IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES

[illegible]

APPEAL BRIEF FOR THE APPELLANT

Ex parte VAN DER VEEN

SYMMETRIC MULTI-PROCESSOR SYSTEM AND METHOD

Serial No. 09/383,115
Appeal No.:
Group Art Unit: 2143

Enclosed is a check in the amount of Three Hundred Forty Dollars ($340.00) to cover the official fee for this Appeal Brief. In the event that there may be any fees due with respect to the filing of this paper, please charge Deposit Account No. 50-2222.

William F. Nixon
Attorney for Appellant(s)
Reg. No. 44,262

SQUIRE, SANDERS & DEMPSEY LLP
8000 Towers Crescent Drive, 14th Floor
Tysons Corner, VA 22182-2700

Atty. Docket: 59119-00004

WFN/cct

Encls: Check No. 012172
Appeal Brief (in triplicate)

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES

In re the Appellant:

Peter H. VAN DER VEEN	Appeal No.:
Serial Number: 09/383,115	Group Art Unit: 2143
Filed: August 25, 1999	Office Action: Joseph E. Alvellino

For: SYMMETRIC MULTI-PROCESSOR SYSTEM AND METHOD

BRIEF ON APPEAL

I. INTRODUCTION

This is an appeal from the final rejection set forth in an Official Action dated April 14, 2004, finally rejecting claims 17-29, all of the claims pending in this application. A Response under 37 CFR § 1.116 was timely filed on July 8, 2004. A Notice of Appeal was timely filed on September 14, 2004, with an appropriate petition for Extension of Time. This Appeal Brief is being timely filed.

II. REAL PARTY IN INTEREST

The real party in interest in this application is QNX Software Systems Ltd., 175 Terence Matthews Crescent, Ottawa, Ontario, Canada K2M 1W8. An assignment from the Inventor to QNX Software Systems Ltd. was recorded by the United States Patent and Trademark Office on August 25, 1999 at reel 010203, frame 0612.

III. STATEMENT OF RELATED APPEALS AND INTERFERENCES

There are no known related appeals and/or interferences which will directly effect or be directly effected by or have a bearing on the Board's decision in this appeal.

IV. STATUS OF CLAIMS

Claims 17-29, all of the claims pending in the present application, are rejected as being unpatentable over certain prior art. Specifically, the rejections of claims 17 and 23-28 as being anticipated by 35 U.S.C. 102(a) over U.S. Patent No. 5,515,538 to Kleiman, claims 18-20 and 29 as being unpatentable under 35 U.S.C. § 103(a) over Kleiman in view of U.S. Patent No. 5,946,487 to Dangelo, and claims 21 and 22 as being unpatentable over Kleiman in view of Dangelo, and further in view of U.S. Patent No. 5,812,844 (Jones et al.), are the subject of this appeal.

V. STATUS OF AMENDMENTS

Claims 1-5 were canceled and claims 6-16 were added in a Preliminary Amendment, filed on February 28, 2000. Claims 6-16 were cancelled and claims 17-29 were added in a Response filed on July 11, 2003. Claims 17-29 are shown in Appendix 1. An Office Action was mailed on April 14, 2004, that rejected claims

17-29, and made the rejections final. An Advisory Action was mailed on August 25, 2004, that upheld the final rejections. No further amendments were filed subsequent to the final Office Action. Therefore, claims 17-29 are pending in the present application.

VI. SUMMARY OF THE INVENTION

The invention relates to computer operating systems for use in a symmetric multiprocessing (SMP) environment. SMP is the processing of application programs and operating systems using multiple processors that share a common operating system and memory. A single copy of the operating system is available to all the processors which share equal access to a common memory via some form of communication path. The goal of an SMP system is to balance the system's workload between the multiple processors, optimising the use of the available resources and providing the User with faster operation. Specification, page 1, lines 20 – 33.

For example, if a network router has four processors, the SMP system will endeavour to distribute the processing load among the processors to increase throughput of network transactions through the router. This aspect will allow computationally intensive routing, like encrypting, for example, to be routed at a higher capacity, and as a result, more data can be routed.

Computer systems, and multiprocessor computer systems in particular, often manage execution of application programs by grouping program steps into "threads." Specification, page 1, lines 9 – 14. In an SMP environment, these threads may run on different processors, even if they are generated by the same application program.

An ideal SMP system would provide a linear performance improvement, such that two processors would execute an application twice as fast as one processor, three processors would execute three times as fast, and so forth. There are, however, critical areas of the Operating System that can only be accessed by one thread at a time because only one Operating System is being shared by all of the processors in the SMP system. If two or more threads are allowed to access the same critical area of the Operating System at the same time, they may interfere with one another causing unexpected or erroneous data to result. Thus, it is necessary to have a mechanism that prevents multiple threads from accessing a critical area of the Operating System at the same time. Specification, page 1, line 26 through page 2, line 7.

To prevent threads from interfering with one another when they call for an operating system service that involves a critical area of the operating system, "locks" are commonly used to limit access to one thread at a time. In such cases, in order to access a critical area of the operating system, the thread must obtain the necessary lock or locks. Once it has completed execution, the thread may then make these locks available to other threads. Two lock handling strategies are described in the specification at page 2, line 8 through page 3, line 23. Referring to Figures 1 and 2, the use of a single global lock, and the use of multiple small locks are shown. Both of these strategies have their respective disadvantages, as described in the specification. In short, they both lock out the entire operating system while the thread holding the lock, executes in its entirety. As shown in Figures 1 and 2, this forces threads to wait for locks to become freed-up by other threads before they can execute. In the single lock system shown in Figure 1, the threads can only execute one at a time, even though multiple processors may be available. There is no improvement in overall processing speed in such a system, regardless of how many additional processors are added. Multiple lock systems as shown in Figure 2 provide some improvement by reducing the locked portions to a defined subset of the operating system. This approach, however, introduces greater complexity and other problems such as deadlocking. Specification, page 2, line 30 through page 3, line 7.

An operating system designed using a microkernel architecture provides greater modularization of its component parts. That is, in a microkernel operating system elements that would normally be integral to the kernel in a monolithic operating system, such as drivers, protocol stacks, and the like, are treated as external processes in the same manner as user applications. This modularization facilitates the definition of operating system calls which execute in non-critical areas, or those parts of the operating system calls that would normally be part of the kernel in a monolithic architecture but are treated as user applications in the microkernel OS, and critical areas, or the remaining elements of the operating system kernel in the microkernel OS. This approach to implementing SMP offers the simplicity of single lock systems to lock the limited function microkernel while realizing the efficiency of multiple lock systems as non-critical areas are not necessarily locked when the microkernel is locked. The modularization of the operating system also avoids deadlocking problems introduced by multiple locks used in monolithic OS SMP implementations.

In other words, the present invention uses a completely different locking strategy than that known in the art. It supports the separation of each operating system call into separate (or multiple) calls to critical and non-critical areas. Locks are obtained only for the call, or calls, to the critical area so that once the call to the critical area has been completed, the lock can be released for use by other operating system calls, as discussed in the specification on page 7 lines 11 – 21, for example, where the "kernel call" is the call to the critical area. Further, the balance of the operating system call, or the portion of the call executing in a non-critical area, can execute without delaying other threads. As shown in Figure 6, this allows the operating system calls to execute much more quickly as far more processing is performed at any given time; that is, the threads are not blocking each other as much, while they are awaiting the release of locks. Adding more processors will allow more threads to execute at the same time, which is in dramatic contrast to that of Figure 1 where the threads execute sequentially.

Of course, this system can only be applied where the operating system calls can actually be broken up into calls to critical and non-critical areas. Typical monolithic operating systems cannot do this because operating system calls are executed internally to the operating system. Specification, page 9, lines 33 – 34. In a monolithic operating system where there is no way to separate an operating system command into the critical and non-critical calls, the entire monolithic operating system is a critical area.

The invention also describes and claims specific implementations which provide further advantages. For example, in the preferred embodiment, the invention is applied to a microkernel operating system design which delegates all functionality to external processes, except for message passing. Specification, page 9, lines 24-28. The microkernel has a single critical area, the microkernel itself, which executes the message passing very quickly, while the external processes are protected by proper thread management. Specification, page 7, lines 17-27. As a result, a single lock may be used to protect the message passing functionality, overcoming the performance problems of the existing SMP strategies.

VII. ISSUES

The issues on appeal are whether claims 17 and 23 – 28 are anticipated under 35 U.S.C. 102 (b) by United States Patent Serial No. 5,515,538 to Kleiman ("the Kleiman patent"), whether claims 18 – 20 and 29 are unpatentable under 35 U.S.C. 103 (a) over the Kleiman patent in view of United States Patent Serial No. 5,946,487 issuing to Dangelo ("the Dangelo patent"), and whether claims 21 and 22 are unpatentable under 35 U.S.C. 103 (a) over the Kleiman patent, in view of the Dangelo patent, and further in view of United States Patent Serial No. 5,812,844 issuing to Jones et al ("the Jones patent"). As discussed below, this Appeal Brief will show that these rejections should be withdrawn, and this application passed to issue.

VIII. GROUPING OF CLAIMS

The Applicant believes that each of the claims is patentably distinct over each of the other claims. However, in the interest of expediency, the Applicant has grouped the claims to reflect the groupings identified by the Office Action in his rejections. That is:

1. claims 17 and 23 – 28 that describe the invention in broad terms;
2. claims 18 – 20 and 29 that include limitations particular to micro-kernel operating systems and functionality particular to micro-kernel operating systems; and

3. claims 21 and 22 that include limitations particular to real-time implementations of the invention.

IX. APPELLANT'S ARGUMENTS

Applicants respectfully submit that each of pending claims 17-29 recites subject matter which is neither disclosed nor suggested by the Kleiman patent, the Dangelo patent, and the Jones patent, either alone or in combination. The arguments that follow are based largely on the arguments filed in response to the final Office Action, but also include arguments that were filed with respect to responses to earlier Office Actions.

Claims 17 and 23-28 Are Not Anticipated by the Kleiman Patent

The Office Action rejected claims 17 and 23-28 under 35 USC 102(b), alleging that these claims are anticipated by the Kleiman patent (United States Patent No. 5,515,538). The rejection is traversed as being based on a reference that neither discloses nor suggests all of the features clearly recited in the independent claims 17, 23, 24, 25 and 26. Claims 27 and 28 depend directly from claim 17.

Claim 17, upon which claims 27 and 28 are dependent, recites a method of symmetric multiprocessing for an inter-process control (IPC) message-passing operating system where operating system calls execute in critical and non-critical areas. The method includes responding to an operating system call requiring access to a critical area of the IPC message-passing operating system by requesting a global lock and responding to the global lock being available by performing the steps of acquiring the global lock, performing the operating system call in the critical area of the IPC message-passing operating system, and releasing the global lock, and responding to the operating system call requiring access to a non-critical area of the IPC message-passing operating system by performing the operating system call in the non-critical area of the IPC message-passing operating system.

Claim 23 recites a computer system. The computer system includes one or more processors. The computer system also includes a memory medium storing an inter-process control (IPC) message-passing operating system where operating system calls execute in critical and non-critical area, in a machine executable form, and a lock manager in a machine executable form. The computer system also includes a communication network interconnecting the one or more processors, and the memory. The lock manager is operable to respond to an operating system call requiring access to a critical area of the IPC message-passing operating by requesting a global lock, and responding to the global lock being available by performing the steps of acquiring the global lock, performing the operating system call in the critical area of the IPC message- passing operating system, and releasing the global lock.

Claim 24 recites an apparatus for symmetric multiprocessing. The apparatus includes an inter-process control (IPC) message-passing operating system means where operating system calls execute in critical and non-critical areas. The apparatus also includes means responsive to an operating system call requiring access to a critical area of the IPC message-passing operating system by requesting a global lock and responding to the global lock being available by performing the steps of acquiring the global lock, performing the operating system call in the critical area of the IPC message-passing operating system, and releasing the global lock.

Claim 25 recites a computer readable memory medium, and storing computer software code executable to perform the steps of responding to an operating system call requiring access to a critical area of an IPC message-passing operating system by requesting a global lock, and responding to the global lock being available by performing the steps of acquiring the global lock, performing the operating system call in the critical area of the IPC message-passing operating system, and releasing the global lock.

Claim 26 recites a computer data signal embodied in a carrier wave. The computer data signal includes a set of machine executable code being executable by a computer to perform the steps of responding to an operating system call requiring access to a critical area of an IPC message-passing operating system by requesting a global lock and responding to the global lock being available by performing the steps of acquiring the global lock, performing the operating system call in the critical area of the IPC message-passing operating system, and releasing the global lock.

As will be discussed below, the cited references of Kleiman, Dangelo and Jones, either alone or in combination, fail to disclose or suggest all of the elements of any of the presently pending claims.

As a matter of background, the Applicant notes that the Kleiman patent discusses the problem of an interrupt causing a current thread context to have to be saved so the interrupt can be serviced. The Kleiman patent does not discuss how operating system calls can be executed more efficiently in an SMP environment, which is the subject matter of the present patent application. The Kleiman patent clearly does not enable the reader to implement an SMP system in the manner of the claimed invention. Thus, the Kleiman patent does not disclose or suggest all the features and cannot possibly be held to anticipate the claims.

The Applicant also wishes to focus on three other major issues in respect of the anticipation rejection in view of the Kleiman patent. Specifically, the Applicant submits that:

1. The Office Action's rejection fails completely if the SunOS cannot be considered to be an inter-process control (IPC) message-passing operating system, having operating system calls which execute in both critical and non-critical areas. If the SunOS is

not such an operating system then the Kleiman patent cannot be considered to disclose or suggest all the features of any of claims 17 or 23–28, because it describes a completely different operating system that has no relevance to the claimed subject matter.

The Applicant will show, hereinafter, that the SunOS is clearly not an operating system which falls into the scope of the claims;

2. the Office Action alleged that claims 17 and 23–28 are anticipated by the Kleiman patent despite the fact that it does not describe all of the steps in the claims, for example, the steps regarding the execution of OS calls in non-critical areas are not disclosed or suggested by the Kleiman patent. The Applicant submits that the test for anticipation has not been satisfied by the Office Action's unsubstantiated allegation that the missing steps are "inherent" as has been done; and

3. the Office Action made reference to column 11 line 39 through column 12 line 11 of the Kleiman patent, as well as column 12 lines 53 – 55, as the basis for the rejection of claim 17. The Applicant submits that these sections of the Kleiman patent are not addressing the claimed symmetric multi processing (SMP) subject matter at all and that the sections are very clearly referring to a manner of handling interrupts. Applicant submits that these portions of the Kleiman patent

cannot be said to anticipate the claims if the portions are not even addressing the same subject matter.

To elaborate on each of these points, Applicant provides the following information regarding distinctions between the Kleiman patent and the pending claims.

On Message-Passing Operating Systems

In response to an earlier Office Action, the Applicant argued that the SunOS *is not* a message-passing operating system. The Office disputed these arguments in the final Office Action at items 14–16, by offering the Powell USENIX publication as evidence that the SunOS *is* a message-passing operating system.

The Applicant has not been able to find any description in the Powell document which discloses or suggests operating system calls to both critical and non-critical areas, as recited in the claims, and the Office Action did not identify any references within the Powell document to support the assertion. Furthermore, while the Powell document does describe the use of kernel threads, the Powell document still clearly describes the SunOS as a monolithic operating system, and does not disclose or suggest a message-passing microkernel operating system.

The SunOS describes a monolithic kernel, so there is no message passing between threads inside the operating system. Microkernel architectures use message passing for operating system threads so the architectures can communicate with one another, while monolithic operating systems do not.

Monolithic operating systems have addressability between all components of the operating system, so message passing between operating system threads is not done.

The Applicant submits that there is ample evidence of the SunOS's characterization as having a monolithic kernel architecture in the industry and provides examples of such evidence in attached Appendix 3. These examples, which were cited in Applicant's Response of July 8, 2004, are discussed below:

1. The article entitled "Toward Real Microkernels" published in the "Communications of the ACM" in Septermber 1996, attached as Tab A in Appendix 3, alone settles the issue completely. This article clearly identifies Solaris as a nonmicrokernel system, in contrast to Mach and Chorus.

The third paragraph in the left column of page 75 states "Some *nonmicrokernel systems* try to reduce communication costs by *avoiding IPC*. As with Chorus and Mach, Solaris and Linux support kernel-loadable modules ... the question of which is superior – kernel-compiler technology or a pure microkernel approach – is open ..." (emphasis added). At this point in the article, the author has already spent five pages explaining how IPC and microkernels operate, generally in the context of Chorus and Mach. Applicant submits that is clear in the quoted paragraph that he is asserting that Solaris and Linux are nonmicrokernel systems that avoid IPC.

Thus, if Solaris avoids IPC, then it cannot disclose or suggest all the features of any of the presently pending claims, and the anticipation rejection is improper.

2. Tab B of Appendix 3 includes notes prepared by Gregory D. Benson, Assistant Professor in the Department of Computer Science at University of San Francisco. These notes are available online at: http://www.cs.usfca.edu/benson/cs326/09-osdesign-notes.pdf

On page 3 of these notes, Solaris is clearly identified as having a monolithic kernel architecture (see item 09-8), while MINIX and Mach are identified as having microkernel architectures (see item 09-10). As noted under item 09-12, "all processes communicate using *message passing*" (emphasis added) for the MINIX OS, and all of the operating system functionality operates as a separate process except for the lowest level functions, such as process management and message passing itself. The balance of the notes elaborate on this aspect.

3. Andy Tanenbaum is a highly respected computer scientist and operating system expert, as well as the designer and implementer of the MINIX microkernel operating system. On his webpage at: http://www.cs.vu.nl/~ast/brown/, he presents an interview he had with Ken Brown, on the development and history of Linux. HTML and text copies of this interview are attached under Tab C of Appendix 3.

Towards the end of this interview, Mr. Tanenbaum explains that QNX and MINIX are examples of microkernel operating systems, while Linux and other similar operating systems are monolithic.

4. A number of searches were done on the Sun website at: http://onesearch.sun.com/ search/developers/index.jsp?qt=solaris&col=javadoc&col=javatecharticles&col=javatutorials&col=devarchive&col=javasc&col=devall, and no evidence could be found of the SunOS or Solaris operating systems being described as anything but monolithic operating systems.

5. Pieter Dumon's article titled "OS Kernels – a little overview and comparison" is attached under Tab D of Appendix 3, and is available online at http://tunes.org/~unios/oskernels.html. On page 3 of the attachment, he indicates that Solaris uses a monolithic architecture, while Mach, L4 and QNX are microkernel architectures.

At the bottom of page 4, he also explains that Mach uses message-passing;

6. Geoffrey Voelker's University of California in San Diego, slides titled "CSE 120 – Principles of Operating Systems – Fall 2001" is attached under Tab E, and is available online at http://www.cs.ucsd.edu/classes/fa01/cse120/lectures/struct-bw.pdf.

Slide 18 identifies Solaris as having a monolithic kernel architecture, while slide 21 identifies Mach and Chorus as examples of microkernel architectures. It is the nature of the microkernel architecture, which is presented in slides 21 and 22, that requires IPC message passing of OS calls. This is in direct contrast to the monolithic architecture shown in slide 16.

7. Under Tab F of Appendix 3, the Applicant has included six pages from the following website: http://cbbrowne.com/info/microkernel.html. Christopher Browne is one of the co-authors of the book "Professional Linux Programming."

Mr. Browne explains that messages and IPC mechanisms are used by microkernel operating systems, such as that of the claimed invention, and are not used by monolithic operating systems, such as that of the SunOS:

a. In the first paragraph of the first page, he identifies the "*messages*" and "interprocess communication (*IPC*)" mechanisms used by microkernel operating systems (emphasis added);

b. In the second paragraph of the first page, he notes that these mechanisms result in modularity and its subsequent advantages which are "not typically found in monolithic or conventional operating systems;"

c. In the third paragraph of the first page, he notes that "microkernels move many of the OS services into "user space" that on other operating systems are kept in the kernel;"

d. Through the course of this document, a number of microkernel operating systems are identified, specifically: Mach, L4 and QNX. There is no mention of the SunOS as a microkernel architecture in this document;

e. At the bottom of the second page, in contrasting microkernel architectures to monolithic architectures, he notes that "communications between components of the extended OS requires that *formalized message-passing mechanisms* be used" (emphasis added).

f. In section 3.1 on the third page, he explains that the "monolithic architecture implements all operating system abstractions in kernel space", while the "microkernel architecture abstracts lower-level OS facilities, implementing them in kernel space, and moves higher-level facilities to processes in user space."

Microkernel architectures require message passing at the OS level because the higher-level facilities are not implemented in the kernel. Monolithic architectures do not require message passing at the OS level because their OS calls are executed in the kernel. The fact that the SunOS monolithic kernel is using OS threads to gain some of the advantages of a microkernel is not going to change this aspect. Thus, SunOS and Solaris, or, basically, different versions of the same operating system, are monolithic kernel-based operating systems. While they may use some operating system threads, they clearly do not use message-passing internal to the operating system. Because they are monolithic, they do not have an inherent separation between critical and non-critical operating areas of the operating system.

On the Test for Anticipation

The Office Action alleged that claims 17 and 23-28 are anticipated by the Kleiman patent, despite the fact that the Kleiman patent does not describe the steps regarding execution of OS calls in non-critical areas. Applicant respectfully traverses and submits that the rejection is improper because the Office does not satisfy the requirements for the anticipation test as the Kleiman patent does not disclose or suggest all the features of the claims.

Applicant notes that the *Manual of Patent Examination Procedure* reads as follows:

> *"A claim is anticipated only if each and every element as set forth in the claim is found, either* ***expressly*** *or* ***inherently*** *described, in a single prior art reference." Verdegaal Bros. v. Union Oil Co. of California, 814 F.2d 628, 631, 2 USPQ2d 1051, 1053 (Fed. Cir. 1987). "The identical invention must*

be shown in as ***complete detail*** *as is contained in the ... claim." Richardson v. Suzuki Motor Co., 868 F.2d 1226, 1236, 9 USPQ2d 1913, 1920 (Fed. Cir. 1989). (emphasis added)*

Clearly, the elements of the claims are not "*expressly*" described in the Kleiman patent, as the Office Action has not even made such an assertion.

The Office Action alleged that a number of the claim elements are "*inherent*" in the Kleiman patent under item 3. For example, the Office Action alleged "it is inherent that other threads can call to non-critical areas of an operating system to execute instructions which are not time-critical." The Office Action also alleged that the Kleiman patent describes the steps regarding the call to "critical" areas of the operating system, but that it is not necessary that it describe the calls to non-critical areas.

Applicant submits that the assertion of inherency is improper. "The fact that a certain result or characteristic <u>may</u> occur or be present in the prior art is not sufficient to establish inherency of that result or characteristic." *In re Rijckaert*, 9 F.3d 1531, 1534, MPEP §2112. The Office Action must "provide a basis in fact and/or technical reasoning to reasonably support the determination that the allegedly inherent characteristic <u>necessarily</u> flows from the teachings of the applied prior art." *Ex parte Levy*, 17 USPQ2d 1461, 1464 (Bd. Pat. App. & Inter. 1990) (emphasis in original). Inherency may not be established by probabilities or possibilities. MPEP §2112. Applicant submits that the claimed features do not necessarily flow from the teachings of the Kleiman patent.

Applicant submits that the claimed features are not "inherent" in the operating system described in the Kleiman patent, or in a typical operating system that operating system calls would require access to both critical and non-critical areas. In order to have a system in which operating system calls may execute in both critical and non-critical areas, the system requires at least the following:

- the functionality to clearly define when calls are going to which areas, and
- the functionality to ensure that there are no consistency problems.

This is very difficult to do, and is the primary reason that most operating systems are designed to execute *only* in critical areas.

Some operating systems do access both critical and non-critical areas, but not in the context of an operating system call. This can be shown in the context of the "garbage collection" example raised by the Office Action.

To begin with, garbage collection is a sub-service. If, when garbage collection is performed, a critical operating system call exists, then, the non-critical garbage collection sub-service will continue to run, causing the operating system call to be blocked, while waiting for the garbage collection to be completed. The operating system call is blocked for consistency. Thus, the operating system call itself, always executes in a critical area and does not execute in a non-critical area.

These arguments apply to all three of the sub-services described by the Office Action.

Typically, monolithic operating systems consider the entirety of a given OS call to be critical, unless it is sleeping, and lock out other calls until they are handled. Most operating systems, and monolithic operating systems in particular, do not have critical and non-critical areas. There is nothing "inherent" about such functionality at all.

Moreover, there is certainly nothing in the Kleiman patent which would teach the reader how to identify OS calls as being directed to critical or non-critical areas, and how to handle them. As noted above, the test requires that "*complete detail*" be provided by the prior art. This certainly has not been done as there is no disclosure or suggestion in the Kleiman patent, on how to do this, or how the features cited by the Office Action necessarily flow from the teachings of the Kleiman patent.

On the Portions of the Kleiman Reference Which Were Cited

In rejecting claim 17, the Office Action made reference to column 11, line 39 through column 12, line 11 of the Kleiman patent, as well as column 12, lines 53–55. Applicant notes that these sections of the Kleiman patent are not addressing the claimed symmetric multi processing (SMP) subject matter at all and that they are clearly referring to a manner of handling interrupts.

As Applicant has noted in previous responses, the Kleiman patent discusses the problem of an interrupt causing a current thread context to have to be saved so the interrupt can be serviced. The Kleiman patent does not disclose or suggest how operating system calls can be executed more efficiently in an SMP environment, which is the subject matter of the presently pending claims. Thus, one skilled in the art would not look to the Kleiman patent for assistance in addressing the problems of the invention because Kleiman is dealing with completely different subject matter, that is improving the efficiency of the handling of interrupts, as opposed to targeting the OS calls.

A review the lines of text cited by the Office Action also bears this out. There is no mention of SMP at all, but a continuous discussion of how to handle interrupts.

The Applicant submits that the lines of text cited by the Office Action certainly do *not* provide the "complete detail" required by the test for anticipation. Thus, every feature of the pending claims is not disclosed or suggested by the Kleiman patent.

The Kleiman patent discusses the problem of an interrupt causing a current thread context to have to be saved, so the interrupt can be serviced. The Kleiman patent does not disclose or suggest how operating system calls can be executed more efficiently in an SMP environment, which is the subject matter of the presently pending claims.

The Kleiman patent describes their interrupt system in the context of Solaris, a monolithic operating system based on a traditional UNIX kernel, which is distinct from the operating system recited in the claims. In a monolithic operating system, all operating system threads are executed as part of the kernel, and are "critical", so there are no "non-critical" areas in the operating system. In contrast, the invention deals with operating system calls which require access to both critical and non-critical areas. Lines 1 - 3 of claim 17, for example, recite an "inter-process control (IPC) message-passing operating system", which is a completely different architecture from that of Kleiman. This wording is reiterated in the body of claim 17 at lines 4 - 5, 9 - 10, 12 - 13 and 14 - 15.

In an IPC message-passing architecture, a typical operating system call requires access to both critical and non-critical areas of the operating system. A single operating system call will access the critical area of the operating system which performs a message passing operation, which passes the message to an external and non-critical software process which executes to perform the balance of the operating system call. As noted above, in monolithic operating systems such as that of the Kleiman patent, all operating system threads are executed as part of the kernel and are "critical", so there are no "non-critical" areas in the Kleiman operating system. Thus, the Kleiman patent does not address the problems that the invention deals with, nor does it disclose or suggest at least these features of the presently pending claims.

One skilled in the art would not look to the Kleiman patent for assistance in addressing the problems of the invention because Kleiman is dealing with completely different subject matter, such as improving the efficiency of their SMP system via handling of interrupts, as opposed to targeting the OS calls. Neither the Kleiman patent nor Solaris have anything to do with the SMP improvement offered by the invention.

Conclusion Regarding Rejection of Claims 17 and 23 - 28

Claims 23-26 include the same limitations as claim 17, but claim the invention in different forms. Specifically, these are system, apparatus, memory medium and signal claims, respectively. These claims are not disclosed or suggested by the Kleiman patent at least for the reasons given above.

Claim 27 depends from claim 17 and includes all of its limitations, but also adds the limitation that the critical area of the message-passing operating system is limited to the message passing operation. As noted above, the Kleiman patent does not describe an operating system which uses internal message-passing, let alone one in which the critical area of the message-passing operating system is limited to the message passing operation. Clearly, the Kleiman patent cannot be said to anticipate claim 27 if there is no mention at all, of this limitation.

Claim 28 depends from claim 27 and includes all of the limitations of claims 17 and 27. It also adds the limitation of a second message-passing operation to complement that of claim 27. Applicant submits that the same argument raised with respect to claim 27 also applies to claim 28.

Further, Applicant notes that the Powell document was raised for the first time in the final Office Action, and submits that this is not appropriate. Applicant submits that he is entitled to at least one attempt to overcome a citation, before a final Action is issued. Applicant did not receive this opportunity.

Applicant also submits that it is improper to rely on two or more references in making a rejection under 35 USC §102(b). In this case, the Office Action has relied on the Kleiman patent and the Powell document in combination. The Powell document does not show that a characteristic not disclosed in the Kleiman patent is inherent, as discussed above. See MPEP §2131.01. Thus, the combination of the Kleiman patent and the Powell document is improper.

Applicant therefore respectfully requests that the anticipation rejection of claims 17 and 23-28 under 35 USC §102(b) be withdrawn.

Claims 18-20 and 29 Are Not Rendered Obvious by the Cited References

The Office Action then rejected claims 18 - 20 and 29 under 35 USC §103(a), alleging that these claims are obvious in view of the Kleiman patent in combination with Dangelo (United States Patent No. 5,946,487), and presumably in view of the Powell document. Applicant respectfully traverses.

To establish a *prima facie* case of obviousness, three basic criteria must be met. First, there must be some suggestion or motivation, either in the references themselves or in the knowledge generally available to one of ordinary skill in the art, to modify the reference or to combine reference teachings. Second, there must be a reasonable expectation of success. Finally, the prior art references must teach or suggest all the claim limitations. MPEP §2142. Applicant submits that a *prima facie* case of obviousness has not been established.

Applicant submits that it is difficult to see how a skilled artisan would be led so clearly to the invention as required by the test for obviousness that the cited references must "*expressly* or *impliedly* suggest the claimed invention" or the Office Action must present a "*convincing* line of reasoning" (emphasis added). Applicant submits that the test has not been satisfied.

As a matter of background, Applicant notes that the title of the Dangelo patent is "Object-Oriented Multi-Media Architecture." The abstract of the Dangelo patent describes the subject matter of the patent as: "An object-oriented, multi-media architecture provides for real-time processing of an incoming stream of pseudo-language byte codes compiled from an object-oriented source program." Applicant submits that Dangelo is so remote from the invention that a skilled artisan would not look to it to deal with the problems of the invention. Dangelo seeks to provide an object-oriented, multi-media architecture that provides for real-time processing. This aspect of Dangelo has nothing to do with the purpose of the invention, which is to improve the performance of symmetric multiprocessing systems by improving the lock-handling process.

The Office Action alleged that Dangelo shows that micro kernel operating systems could be used in the application of the invention, but this argument fails for a number of reasons. Most important is that Dangelo's definition of a "micro kernel operating system" is far different than the definition generally held in the art, and his definition certainly does not fall within that outlined in the specification. See page 6, line 34 through page 7, line 2 which reads that "A micro kernel operating system is one in which the operating system itself provides minimal services [which] delegate the usual operating system functions to external processes." At lines 35 - 43 of column 9 of the Dangelo patent, the reference clearly defines his "micro kernel" operating system as what is known as a "monolithic" operating system, as it does far more internal processing. "The micro-kernel also attends to handling the network file system (NFS), networking operations, peripheral device drivers, virtual memory management, user interface, and other tasks for which the operating system conventionally is responsible." This is simply not a micro-kernel operating system as known in the art, and as defined in the specification. Because Dangelo handles all of this processing internally, it does not have the functionality characteristic of micro kernel operating systems.

The Dangelo patent describes mutual exclusion locks or mutexes for the protection of object-oriented multimedia data. However, mutexes have been known in the art for many years. Mutexes are standard POSIX thread-level synchronization primitives, like the semaphores addressed in response to an earlier Office Action. Dangelo's mutexes are used to ensure exclusive access to data shared between threads and not to protect operating system calls. If the Dangelo mutexes were used to protect OS calls, they would block all of the OS functionality during an OS call, which Dangelo defined at lines 35 - 43 of column 9 as "handling the network file system (NFS), networking operations, peripheral device drivers, virtual memory management, user interface, and other tasks." Clearly, the techniques of Dangelo do little to assist in developing an effective SMP system as claimed.

Redesigning an operating system to use mutex-like functionality for operating system calls would still not lead one to the claimed invention. The claimed invention lies in the strategic use of locks for only part of the operating system call, that is, the part within the critical areas of the operating system call. This is what provides the ability to overlap non-critical areas of operating system calls as shown in Figure 6 of the subject patent application.

While Dangelo may describe an application of mutexes, the reference clearly does not describe any of the functionality necessary to implement the invention of claim 18, either independently or in view of the Kleiman and Powell references.

Thus, claim 18 cannot be considered obvious in view of the cited combination of references. The Kleiman patent does not disclose or suggest all of the limitations of the parent claim, claim 17, and the Dangelo patent does not disclose or suggest the missing limitations. Further, neither of these references these additional limitations of claim 18.

Applicant submits that claims 19, 20 and 29 distinguish over the cited references in the same manner as described above with regard to claim 18. Applicant therefore respectfully requests that the obviousness rejection of claims 18–20 and 29 be withdrawn.

Claims 21 and 22 Are Not Rendered Obvious by the Cited References

The Office Action also rejected claims 21 and 22 in view of the Kleiman patent in combination with Dangelo and Jones et al. (United States Patent No. 5,812,844), and presumably in combination with the Powell document. Applicant respectfully traverses.

Applicant notes that claims 21 and 22 depend from claims 17, 18, 19 and 20. Therefore, claims 21 and 22 include all of the limitations of claims 17 - 20. The Jones patent does nothing to address the limitations of claims 17 - 20, with respect to the limitations that the Kleiman and Dangelo patents are lacking. Because the Kleiman, Dangelo and Jones patents, either alone or in combination, do not disclose or suggest all the limitations of the parent claims, Applicant submits that the dependent claims cannot be considered obvious in view of the same combination of references.

The Jones patent describes a method of scheduling thread execution, but it does not disclose or suggest such a method in the context of a pre-emptable micro kernel environment as per claim 19, let alone one in which operating system calls execute in both critical and non-critical areas as per claim 17. The Jones patent describes how threads would be scheduled in a completely different architecture, so its teachings would be of essentially no value in the context of claims 21 and 22.

Another important issue which the Office Action did not address is how the limitations cobbled together from the Kleiman, Dangelo and Jones patents and the Powell document would actually be combined into a working program, or have any expectation of success. Software processes can always be described in terms of a small number of discrete steps, such as storing, transmitting, calculating and receiving. Taken individually, these discrete steps do not teach the reader anything, as almost every program has them. Only when these steps are taken in the context of a patent claim as a whole, do they accomplish the claimed features. Citing one reference that describes locks, and another that describes micro kernel operating systems, does not disclose or suggest how the reader is to incorporate locks into a micro kernel operating system. It is therefore improper to allege that a claim is obvious simply because individual references can be found which recite certain elements of the claim in isolation.

Applicant submits that there is no instruction as to how to combine the elements of the Kleiman, Dangelo and Jones patents, let alone any motivation or suggestion to seek out these patents and arrive at the claimed invention.

Applicant submits that claims 21 and 22 are not disclosed or suggested by the cited combination of references, and therefore respectfully requests that the obviousness rejection of these claims be withdrawn.

ADVISORY ACTION

In the Advisory Action which was mailed by the USPTO on August 25, 2004, the Office again rejected all of Applicant's arguments. Applicant does not agree with the Office's assertions at all.

Firstly, the Office Action asserted that "the term IPC message passing operating system ... merely means that the processes have some medium in which to communicate between processes." Applicant does not agree.

Applicant submits that the term is well know this assertion in the art, and has a far more specific interpretation than the one the Office has given it. The Office has never produced any evidence to support this assertion. Applicant, on the other hand, has defined the term in the specification, and clearly communicated what he seeks to protect via the description, claims, figures and file history as a whole.

Secondly, the Office returned to the comparison to mutexes, which Applicant responded to earlier in prosecution. As previously noted, mutexes are well known in the art and have nothing to do with the claimed invention or the implementation of an effective SMP system. As noted above with respect to the Dangelo reference, redesigning an operating system to use mutex-like functionality for operating system calls would still not disclose or suggest the claimed invention. The invention lies in the strategic use of locks for only part of the operating system call, such as the part within the critical areas of the operating system call. There is no disclosure or suggestion at all in the area of mutex implementations to suggest such an application.

X. CONCLUSION

For all of the above noted reasons, it is strongly submitted that certain clear and patentable differences exist between the present invention as claimed in claims 17-29, and the cited references relied upon by the Office Action. This final rejection is in error, therefore, it is respectfully requested that this Honorable Board of Patent Appeals and Interferences reverse the Office Action's decision in this case regarding the rejection of claims 17-29, and indicate the allowability of all of pending claims 17-29.

In the event that this paper is not being timely filed, the applicants respectfully petition for an appropriate extension of time. Any fees for such an extension together with any additional fees which may be due with respect to this paper may be charged to Counsel's Deposit Account 50-2222.

Respectfully submitted,

SQUIRE, SANDERS & DEMPSEY LLP

[illegible]

William F. Nixon
Attorney for Applicant
Registration No. 44,262

Atty. Docket No.: 59119.00004

8000 Towers Crescent Drive, 14th Floor
Tysons Corner, VA 22182-2700
Tel: (703) 720-7800
Fax (703) 720-7802

WFN:cct

Encls: Appendices 1-3

APPENDIX 1

CLAIMS ON APPEAL

Claims 1-16 (Cancelled).

17. A method of symmetric multiprocessing for an inter-process control (IPC) message-passing operating system where operating system calls execute in critical and non-critical areas, said method comprising the steps of:

responding to an operating system call requiring access to a critical area of said IPC message-passing operating system by:

requesting a global lock; and

responding to said global lock being available by performing the steps of:

acquiring said global lock;

performing said operating system call in said critical area of said IPC message-passing operating system; and

releasing said global lock; and

responding to said operating system call requiring access to a non-critical area of said IPC message-passing operating system by:

performing said operating system call in said non-critical area of said IPC message-passing operating system.

18. A method as claimed in claim 17, wherein said IPC message-passing operating system includes a micro kernel operating system and wherein:

said step of responding to an operating system call requiring access to a critical area of said IPC message-passing operating system, includes the step of responding to an operating system call requiring access to a critical area of said micro kernel operating system;

said step of performing said operating system call in said critical area of said IPC message-passing operating system, includes the step of performing said operating system call in said critical area of said micro kernel operating system;

said step of responding to said operating system call requiring access to a non-critical area of said IPC message-passing operating system includes the step of responding to said operating system call requiring access to a non-critical area of said micro kernel operating system; and

said step of performing said operating system call in said non-critical area of said IPC message-passing operating system includes the step of performing said operating system call in said non-critical area of said micro kernel operating system.

19. The method as claimed in claim 18, wherein said micro kernel operating system includes a pre-emptable micro kernel operating system, said method further comprising the steps of:

pre-empting any non-critical threads currently executing on said pre-emptable micro kernel operating system prior to said step of acquiring said global lock; and

reinstating said pre-empted threads following said step of releasing said global lock.

20. The method as claimed in claim 19, wherein said step of performing said operating system call to said critical area comprises the steps of:

locking said critical area of said pre-emptable micro kernel operating system;

entering said critical area of said pre-emptable micro kernel operating system;

executing operating system functions as required; and

exiting said critical area of said pre-emptable micro kernel operating system.

21. The method as claimed in claim 20, further comprising the step of prioritizing execution of threads in accordance with how their respective call latencies will impact real time operation.

22. The method as claimed in claim 20, wherein said operating system includes a real time operating system, and said method further comprises the step of scheduling execution of said threads to be performed by predetermined time deadlines.

23. A computer system comprising:

one or more processors;

a memory medium storing an inter-process control (IPC) message-passing operating system where operating system calls execute in critical and non-critical areas, in a machine executable form, and a lock manager in a machine executable form;

a communication network interconnecting said one or more processors, and said memory; and

said lock manager being operable to:

respond to an operating system call requiring access to a critical area of said IPC message-passing operating system by:

requesting a global lock; and

responding to said global lock being available by performing the steps of:

acquiring said global lock;

performing said operating system call in said critical area of said IPC message-passing operating system; and

releasing said global lock.

24. An apparatus for symmetric multiprocessing comprising:

an inter-process control (IPC) message-passing operating system means where operating system calls execute in critical and non-critical areas;

means responsive to an operating system call requiring access to a critical area of said IPC message-passing operating system by:

requesting a global lock; and

responding to said global lock being available by performing the steps of:

acquiring said global lock;

performing said operating system call in said critical area of said IPC message-passing operating system; and

releasing said global lock.

25. A computer readable memory medium, storing computer software code executable to perform the steps of:

responding to an operating system call requiring access to a critical area of an IPC message-passing operating system by:

requesting a global lock; and

responding to said global lock being available by performing the steps of:

acquiring said global lock;

performing said operating system call in said critical area of said IPC message-passing operating system; and

releasing said global lock.

26. A computer data signal embodied in a carrier wave, said computer data signal comprising a set of machine executable code being executable by a computer to perform the steps of:

responding to an operating system call requiring access to a critical area of an IPC
message-passing operating system by:

requesting a global lock; and

responding to said global lock being available by performing the steps of:

acquiring said global lock;

performing said operating system call in said critical area of said IPC message-passing operating system; and

releasing said global lock.

27. The method as claimed in claim 17, where said critical area of said IPC message-passing operating system is limited to the message passing functionality of said IPC message-passing operating system, and wherein said step of performing said operating system call in said critical area of said operating system comprises the step of:

performing an IPC message-pass operation for said operating system call.

28. The method as claimed in claim 27, where said IPC message passing operating system requires a message-pass before and after execution of said operating system call in said non-critical area of said IPC message-passing operating system, said method comprising the subsequent steps of:

requesting a global lock a second time; and

responding to said global lock being available by performing the steps of:

acquiring said global lock a second time;

performing a second message-pass operation for said IPC message-passing operating system call; and

releasing said global lock a second time.

29. The method as claimed in claim 17, wherein said IPC message-passing operating system includes a micro kernel operating system having operating system calls executing in external processes, and wherein said step of performing said operating system call in said non-critical area of said IPC message-passing operating system comprises the step of:

performing said external process for said operating system call.

APPENDIX 2

DRAWINGS OF APPLICATION SERIAL NO. 09/882,949

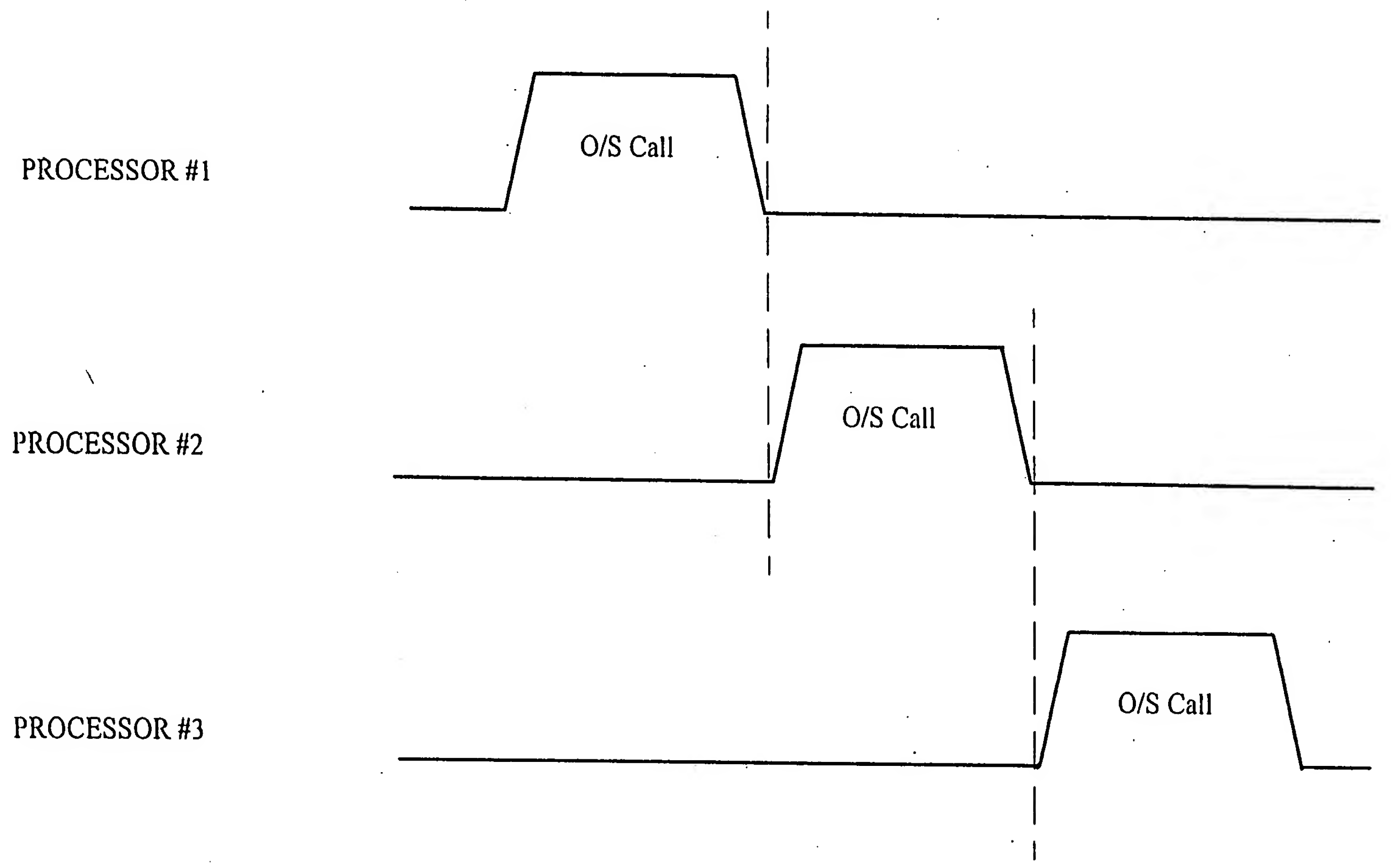


FIGURE 1 - PRIOR ART

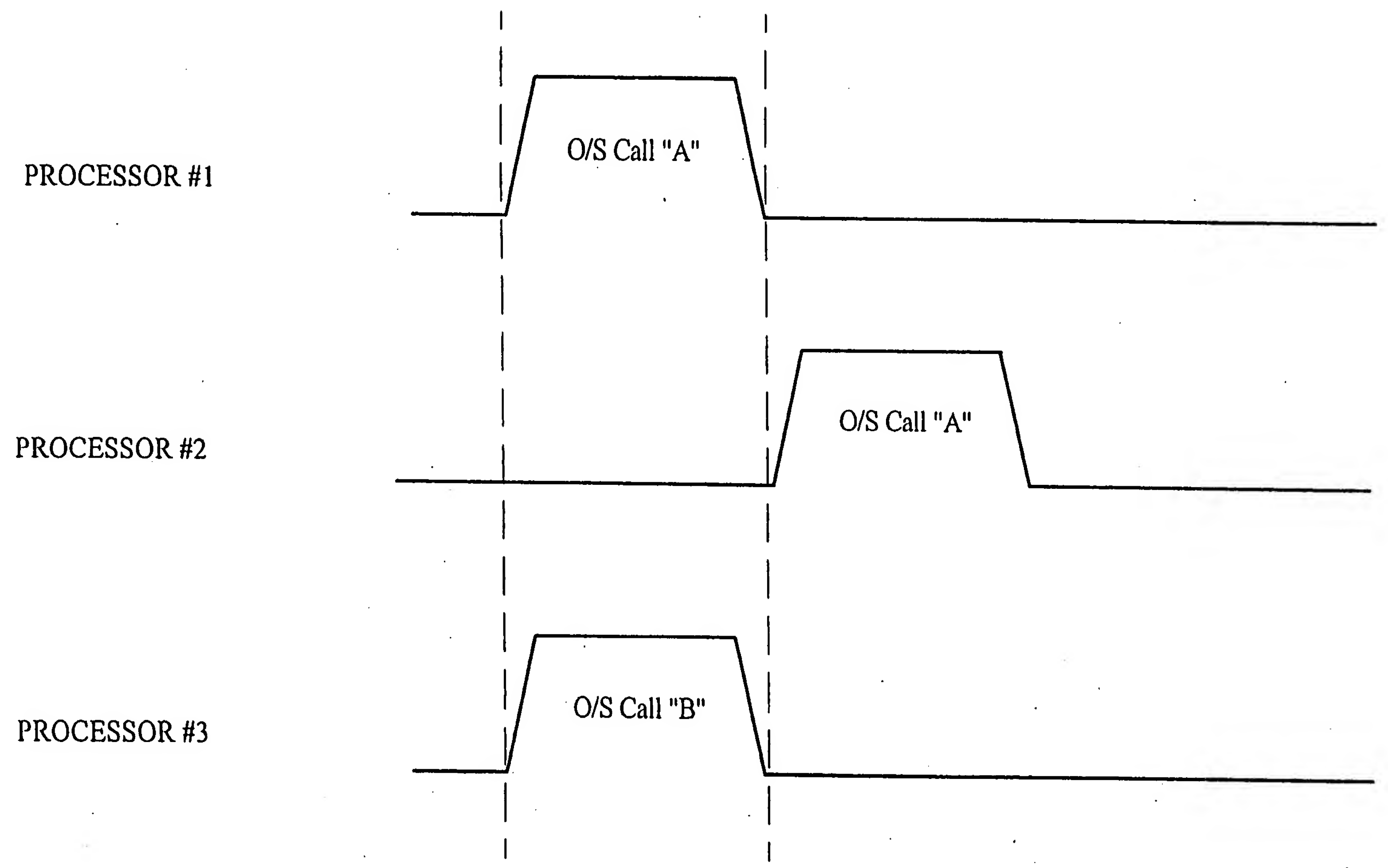


FIGURE 2 - PRIOR ART

FIGURE 3

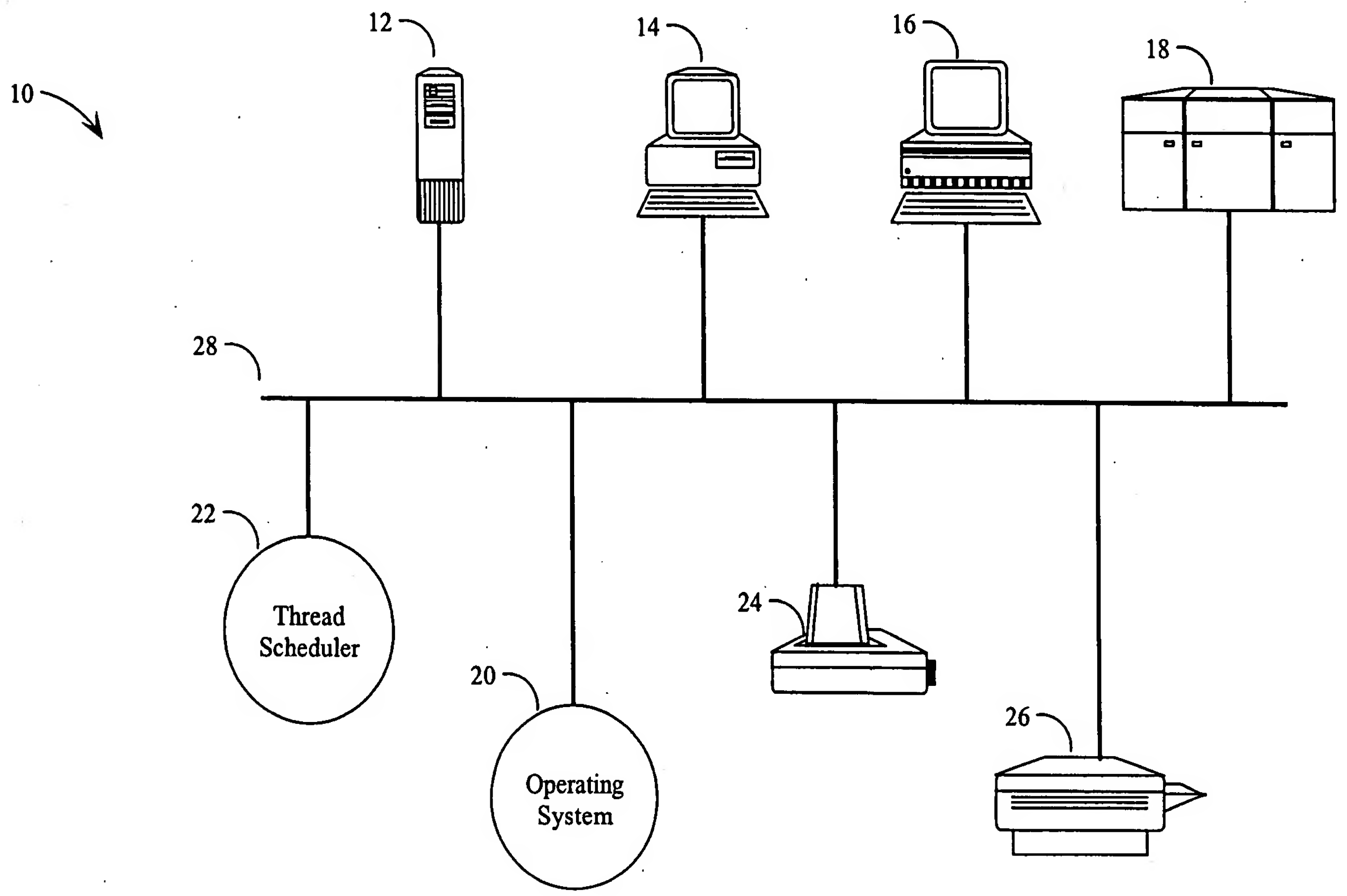

10
12
14
16
18
28
22
Thread Scheduler
20
Operating System
24
26

FIGURE 4

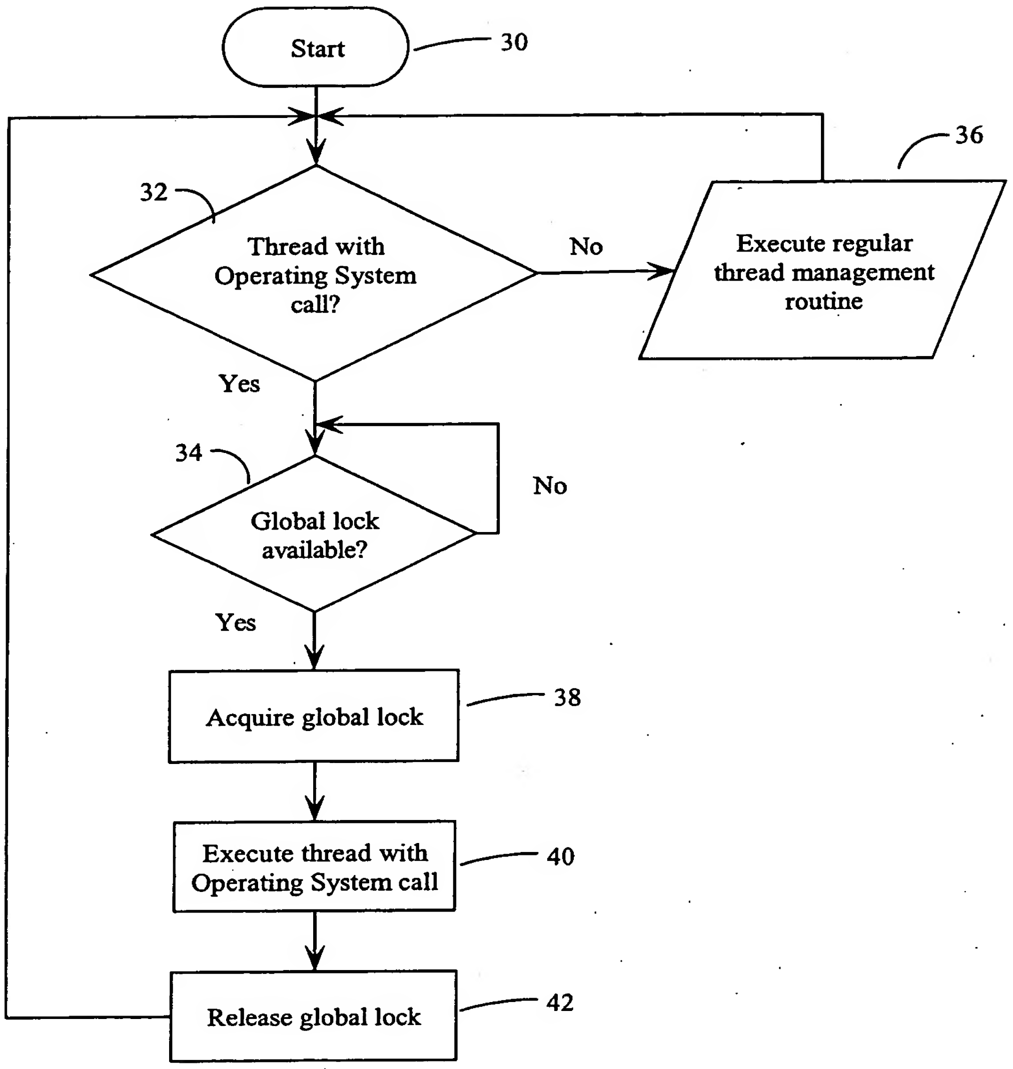
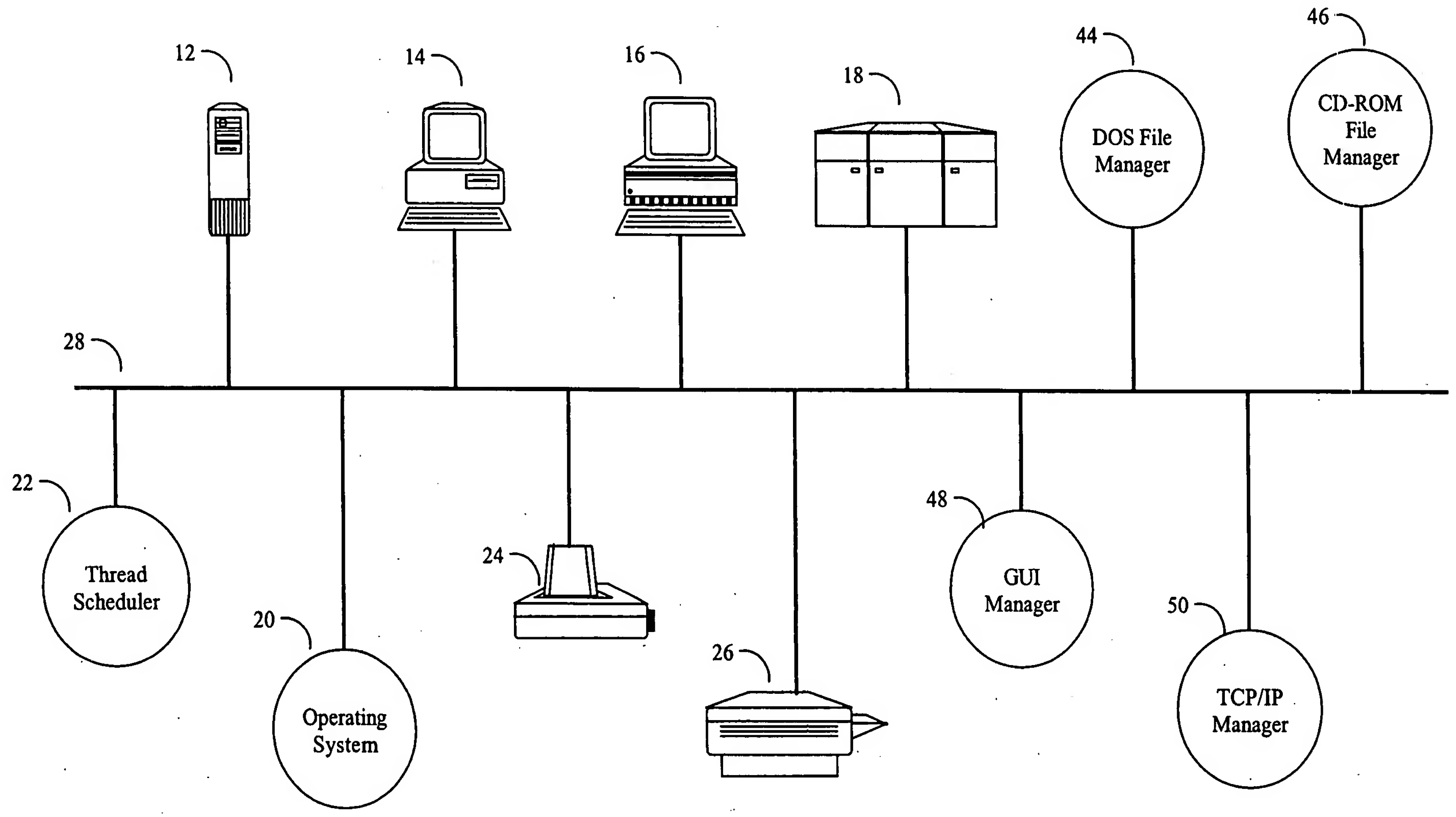

Start
30
32
Thread with Operating System call?
No
Execute regular thread management routine
36
Yes
34
Global lock available?
No
Yes
Acquire global lock
38
Execute thread with Operating System call
40
Release global lock
42

FIGURE 5


12
14
16
18
44
DOS File Manager
46
CD-ROM File Manager
28
22
Thread Scheduler
20
Operating System
24
26
48
GUI Manager
50
TCP/IP Manager

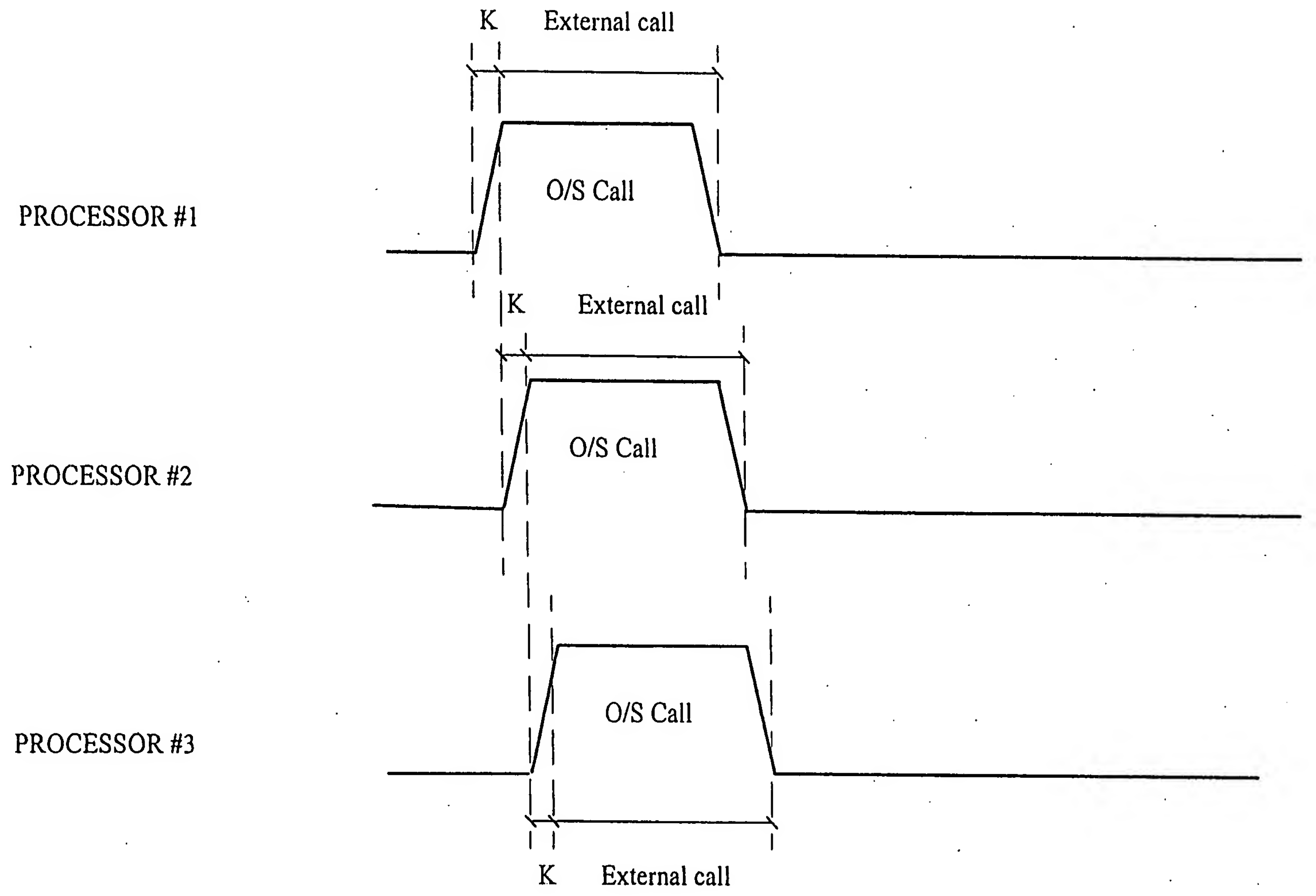
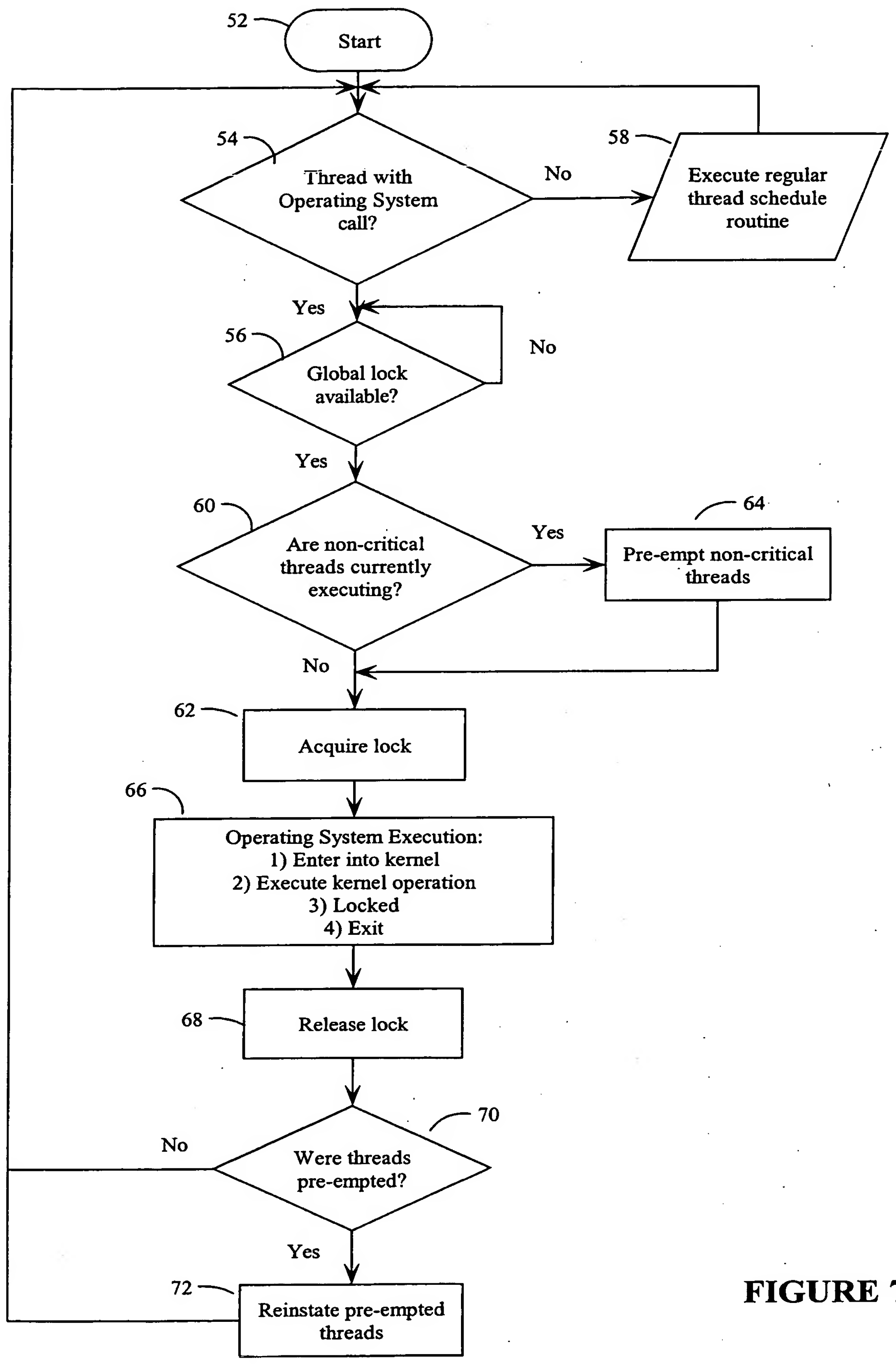

K
External call
PROCESSOR #1
O/S Call
K
External call
PROCESSOR #2
O/S Call
PROCESSOR #3
O/S Call
K
External call

FIGURE 6



52
Start
54
Thread with Operating System call?
No
58
Execute regular thread schedule routine
Yes
56
Global lock available?
No
Yes
60
Are non-critical threads currently executing?
Yes
64
Pre-empt non-critical threads
No
62
Acquire lock
66
Operating System Execution:
1) Enter into kernel
2) Execute kernel operation
3) Locked
4) Exit
68
Release lock
70
Were threads pre-empted?
No
Yes
72
Reinstate pre-empted threads

FIGURE 7

APPENDIX 3

TOWARD REAL MICROKERNELS

The inefficient, inflexible first generation inspired development of the vastly improved second generation, which may yet support a variety of operating systems.

THE microkernel story is full of good ideas and blind alleys. The story began with enthusiasm about the promised dramatic increase in flexibility, safety, and modularity. But over the years, enthusiasm changed to disappointment, because the first-generation microkernels were inefficient and inflexible. • Today, we observe radically new approaches to the microkernel idea that seek to avoid the old mistakes while overcoming the old constraints on flexibility and performance. The second-generation microkernels may be a basis for all types of operating systems, including timesharing, multimedia, and soft and hard real time.

The Kernel Vision

Traditionally, the word kernel denotes the mandatory part of the operating system common to all other software. The kernel can use all features of a processor (e.g., programming the memory management unit); software running in user mode cannot execute such safety-critical operations.

Most early operating systems were implemented by means of large monolithic kernels. Loosely speaking, the complete operating system—scheduling, file system, networking, device drivers, memory management, paging, and more—was packed into a single kernel.

In contrast, the microkernel approach involves minimizing the kernel and implementing servers outside the kernel. Ideally, the kernel implements only address spaces, interprocess communication (IPC), and basic scheduling. All servers—even device drivers—run in user mode and are treated exactly like any other application by the kernel. Since each server has its own address space, all these objects are protected from one another.

When the microkernel idea was introduced in the

Jochen Liedtke

late 1980s, the software technology advantages seemed obvious:

- Different application program interfaces (APIs), different file systems, and perhaps even different basic operating system strategies can coexist in one system. They are implemented as competing or cooperating servers.
- The system becomes more flexible and extensible. It can be more easily and effectively adapted to new hardware or new applications. Only selected servers need to be modified or added to the system. In particular, the impact of such modifications can be restricted to a subset of the system, so all other processes are not affected. Furthermore, modifications do not require building a new kernel; they can be made and tested online.
- All servers can use the mechanisms provided by the microkernel, such as multithreading and IPC.
- Server malfunction is as isolated as normal application malfunction.
- These advantages also hold for device drivers.
- A clean microkernel interface enforces a more modular system structure.
- A smaller kernel can be more easily maintained and should be less prone to error.
- Interdependencies between the various parts of the system can be restricted and reduced. In particular, the trusted computing base (TCB) comprises only the hardware, the microkernel, a disk driver, and perhaps a basic file system.[1] Other drivers and file and network systems need no longer be absolutely trustworthy.

Although these advantages seemed obvious, the first-generation microkernels could not substantiate them.

The First Generation

The microkernel idea met with efforts in the research community to build post-Unix operating systems. New hardware (e.g., multiprocessors, massively parallel systems), new application requirements (e.g., security, multimedia, and real-time distributed computing) and new programming methodologies (e.g., object orientation, multithreading, persistence) required novel operating-system concepts.

The corresponding objects and mechanisms—threads, address spaces, remote procedure calls (RPCs), message-based IPC, and group communication—were lower-level, more basic, and more general abstractions than the typical Unix primitives. In addition to the new mechanisms, providing an API compatible with Unix or another conventional operating system was a sine qua non; hence implementing Unix on top of the new systems was a natural consequence. Therefore, the microkernel idea became widely accepted by operating-system designers for two completely different reasons: (1) general flexibility and power and (2) the fact that microkernels offered a technique for preserving Unix compatibility while permitting development of novel operating systems.

Many academic projects took this path, including Amoeba [19], Choices [4], Ra [1], and V [7]; some even moved to commercial use, particularly Chorus [11], L3 [15], and Mach [10], which became the flagship of industrial microkernels.

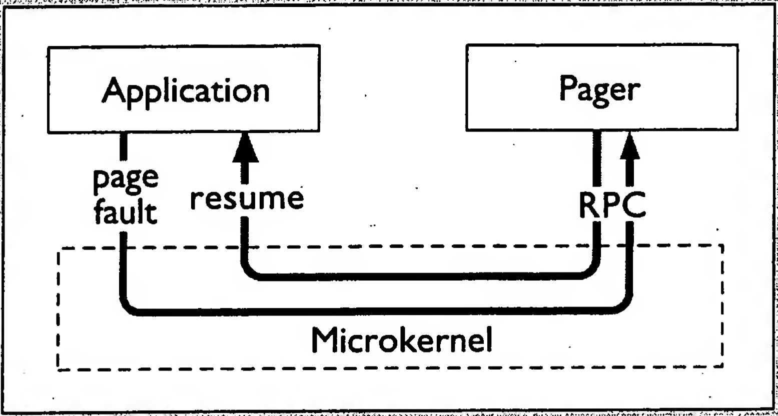


Figure 1. Page fault processing

Mach's external pager [22] was the first conceptual breakthrough toward real microkernels. The conceptual foundation of the external pager is that the kernel manages physical and virtual memory but forwards page faults to specific user-level tasks. These pagers implement mapping from virtual memory to backing store by writing back and loading page images. After a page fault, they usually return the appropriate page image to the kernel, which then establishes the virtual-to-physical memory mapping (see Figure 1). See the sidebar "Frequently Asked Questions on External Pagers."

This technique permits the mapping of files and databases into user address spaces without having to integrate the file/database systems into the kernel. Furthermore, different systems can be used simultaneously. Application-specific memory sharing and distributed shared memory can also be implemented by user-level servers outside the kernel.

The second conceptual step toward microkernels was the idea of handling hardware interrupts as IPC

[1]The TCB is the set of all components whose correct functionality is a precondition for security. Hardware and kernel both belong to the TCB.

Frequently Asked Questions on External Pagers

Is a pager required inside the microkernel? No.

How do user-level pagers affect system security? A pager can corrupt the data it maintains. So you rely on the correct functionality of the pager you use, whether it is kernel-integrated or user-level. However, different noninterfering user-level pagers can be used to increase security by, for example, holding sensitive data in a trustworthy and stable (standard) pager and less critical data in potentially less trustworthy pagers. Note that semantic dependencies are similar on all levels and are not operating-system specific; users rely as much on the correct functionality of a compiler as on that of a database system.

How expensive are user-level pagers? In principle, the overhead compared to that of an integrated pager is only one IPC and should be negligible. In practice, however, most first-generation microkernels use a complicated protocol with up to eight additional IPCs and implement IPC inefficiently. Overheads of 1,000 μs (Digital Equipment Corp. Alpha, 133 MHz) can occur. A microkernel of the second generation solves this performance problem by using only one additional IPC and making it fast (less than 10 μs).

messages [17] and including I/O ports in address spaces. The kernel captures the interrupt but does not handle it, instead generating a message for the user-level process currently associated with the interrupt. Thus, interrupt handling and device I/O are done completely outside of the kernel in the following way:

```
driver thread:
  do
    wait for (msg, sender) ;
    if sender = my hardware interrupt
      then read/write i/o ports ;
           reset hardware interrupt
      else . . .
    fi
  od .
```

In this approach, device drivers can be replaced, removed, or added dynamically—without linking a new kernel and rebooting the system. Drivers can thus be distributed to end users independent of the kernel. Furthermore, device drivers profit from using such microkernel mechanisms as multithreading, IPC, and address spaces. See the sidebar "Frequently Asked Questions on User-Level Device Drivers."

Disappointments

An appealing concept is only one side of the coin; the other is usefulness. For example, are the concepts flexible enough and the costs of that flexibility low enough for such real-world problems as multimedia, real time, and embedded systems?

With respect to efficiency, the communication facility is the most critical microkernel mechanism. Each invocation of an operating system or application service requires an RPC, generally consisting of two IPCs—the call and the return message. Therefore, microkernel architects spent much time optimizing the IPC mechanisms. Steady progress yielded up to twofold improvement in speed, but by 1991, the steps became less and less effective. Mach 3 stabilized at about 115 μs per IPC on a 486-DX50—comparable to most other microkernels. For example, a conventional Unix system call—roughly 20 μs on this hardware—has about 10 times less overhead than the Mach RPC. It seemed that 100 μs was the inherent cost of an IPC and that the concept had to be evaluated on this basis.

Since absolute time lacks meaning on its own, two more practical criteria are used for evaluation:

- Applications must not be degraded by the microkernel. This conservative criterion is a necessary precondition for practical acceptance.
- Microkernels must efficiently support new types of applications that cannot be implemented with good performance on conventional monolithic ker-

Frequently Asked Questions on User-Level Device Drivers

Can you really use a user-level disk driver for demand paging? Yes.

Can a user-level driver corrupt the system? Drivers are encapsulated in address spaces, so they can access only the memory and the I/O ports granted to them. From this point of view, they can corrupt any component relying on their functionality, but not the whole system. However, drivers control hardware, and if this hardware permits system corruption (e.g., by physical memory access via DMA), no kernel can prevent the driver from corrupting the system. Risk depends on the hardware accessible to the driver.

Do user-level drivers increase security? Yes, because of encapsulation. For example, a mouse driver can do no more harm than an editor.

How expensive are user-level drivers? Most first-generation kernels (except L3) implement all time-critical drivers as kernel drivers. However, due to the fast communication facilities of second-generation microkernels, user-level drivers perform as well as integrated drivers in these systems. Less than 10 μs are required per interrupt or driver RPC.

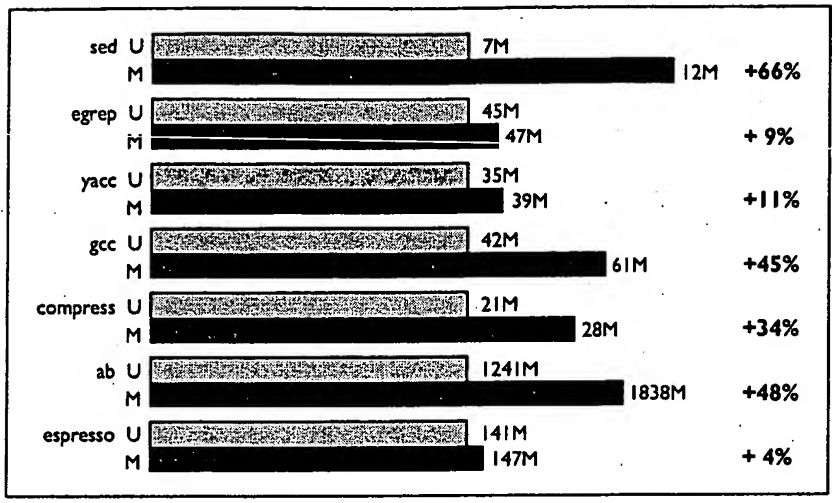


Figure 2. Non-idle cycles under Ultrix and Mach

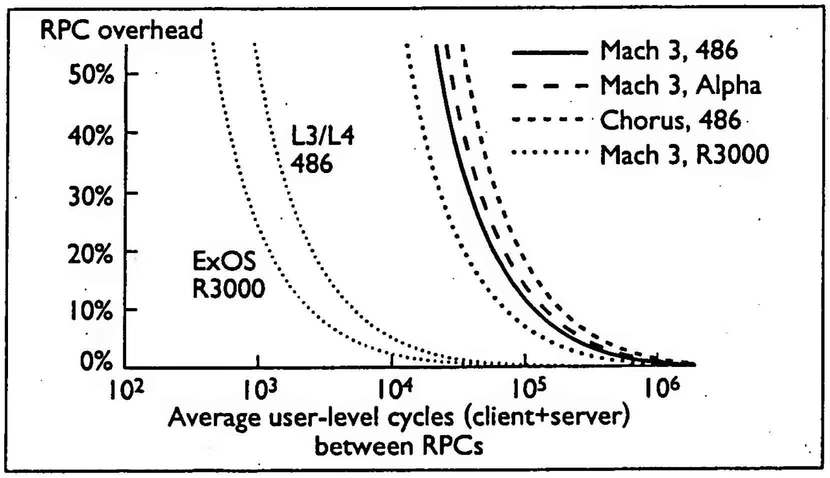


Figure 3. Relative RPC overhead

nels. This progressive criterion must be satisfied for a real advance in operating-system technology.

The conservative criterion can be evaluated by benchmarks running on the same hardware platform under a monolithic and a microkernel-based operating system. This method measures not only the primary (direct) IPC costs but also the secondary costs induced by structuring software with the client/server paradigm and by using the IPC mechanism.

Some applications performed as well under a microkernel as under a monolithic kernel, and a few slightly better. Unfortunately, other applications were substantially degraded. Chen and Bershad [6] compared applications under Ultrix and Mach on a DECStation 5200/200 and found peak degradations of up to 66% on Mach (compared to Ultrix) (see Figure 2). Condict, et al [8] compared an eight-user AIM III benchmark on a 486-DX50 under a monolithic OSF/1 and a Mach-based OSF/1 and measured an average 50% degradation. The measurements corroborate that the degradation is essentially caused by IPC: At least 73% of the measured penalty is related to IPC or activities that are its direct consequence; 10% comes from multiprocessor provisions that could be dropped on this uniprocessor; and the remaining 17% is due to unspecified reasons.

Chen found that the performance differences are caused in part by a substantially higher cache-miss rate for the Mach-based system. This result could point to a principal weakness of server architectures when used with microkernels. However, the increased cache misses are caused by the Mach kernel, invoked for IPC, not by the higher modularity of the client/server architecture.

The measured microkernel penalty is too large to be ignored. From a practical point of view, the pure microkernel approach of the first generation was a failure.

As a consequence, both Chorus and Mach reintegrated the most critical servers and drivers into the kernel [3, 8]. Chorus's "supervisor actors" and Mach's "kernel-loaded tasks" run in kernel mode, can freely access kernel space, and can freely interact with each other. Gaining performance by saving user-kernel and address-space switches looked reasonable and seemed successful. However, while solving some performance problems, this workaround weakens the microkernel approach. If most drivers and servers are included in the kernel for performance reasons, the benefits—encapsulation, security, and flexibility—largely disappear.

Evaluation of the progressive criterion must be based on upcoming trends and applications. Object-orientation and distribution will cause increased cross-address-space interaction. As a consequence, RPC granularity will become finer; on average, clients and server will spend fewer cycles between successive RPCs.

Figure 3 shows the relative RPC overhead as a function of the average number of cycles spent by client and server between two successive RPCs. The overhead is given with respect to an ideal system where RPC is free. Total overhead means a program takes twice as much time for execution due to RPC as in the ideal system. It seems reasonable to assume that 10% is tolerated in most cases but that 50% is not. The 10% limit allows applications of first-generation microkernels to use one RPC every 100,000 user-level cycles; roughly 10,000 lines of client and server code must be executed before invoking the next

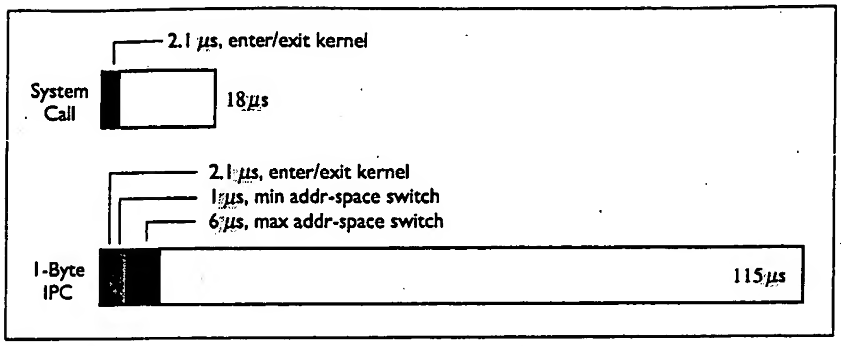


Figure 4. Hardware (black) vs. Mach (black and white) costs when used with a 486-DX50 CPU

RPC. The disappointing conclusion is that first-generation microkernels do not support fine-grained use of RPC. For comparison, Figure 3 also shows overheads for two second-generation microkernels. Under the 10% restriction, the second-generation microkernels permit roughly one RPC per 400 lines of executed user-level code.

Beside this performance-based inflexibility, another shortcoming became apparent over the years. The external-pager concept is in principle not sufficiently flexible. Its most important technical weakness is that main memory is still managed by the microkernel and can be controlled only rudimentarily by the external pager. However, multimedia file servers, real-time applications, frame buffer management, and some nonclassical applications require complete main-memory control.[2]

Conceptually, the weakness is the policy inside the microkernel. A "policy interface" permitting paramaterization and tuning of a built-in policy is convenient as long as that policy is suited for all applications. Its limitations are obvious as soon as a really novel policy is needed or a substantial modification is needed for a predefined policy.

The Second Generation

The deficiency analysis of the early microkernels identified user-kernel-mode switches, address-space switches, and memory penalties as primary sources of disappointing performance. Regarded superficially, this analysis was correct, because it was supported by detailed performance measurements.

Surprisingly, a deeper analysis shows that the three points—user-kernel-mode switches, address-space switches, and memory penalties—are not the real problems; the hardware-inherited costs of mode and address-space switching are only 3%–7% of the measured costs (see Figure 4). A detailed discussion can be found in [16].

The situation was strange. On the one hand, we knew the kernels could run at least 10 times faster; on the other, after optimizing microkernels for years, we no longer saw new significant optimization possibilities. This contradiction suggested the efficiency problem was caused by the basic architecture of these kernels.

Indeed, most early microkernels evolved step by step from monolithic kernels, remaining rich in concepts and large in code size. For example, Mach 3 offers approximately 140 system calls and needs more than 300 Kbytes of code. Reducing a large monolithic kernel may not lead to a real microkernel.

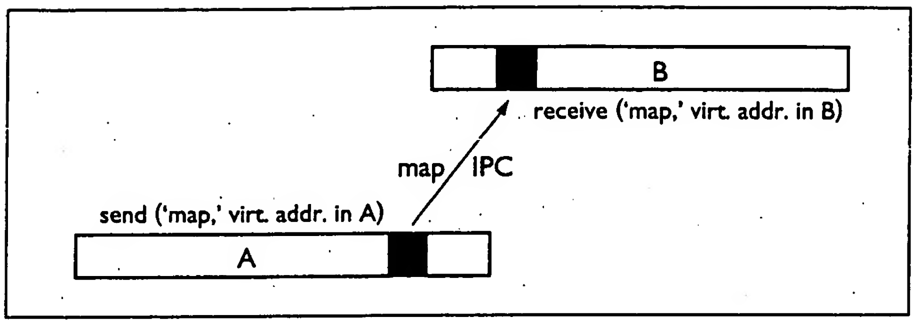
Radical New Designs

A new radical approach, designing a microkernel architecture from scratch, seemed promising and necessary. Exokernel [9] and L4 [16], discussed here, both concentrate on a minimal and clean new architecture and support highly extensible operating systems.

- **Exokernel.** Exokernel, developed at MIT in 1994-95, is a small, hardware-dependent microkernel based on the idea that abstractions are costly and restrict flexibility [9]. The microkernel should multiplex hardware primitives in a secure way. The current exokernel, which is tailored to the Mips architecture and gets excellent performance for kernel primitives, is based on the philosophy that a kernel should provide no abstractions but only a minimal set of primitives (although the Exokernel includes device drivers). Consequently, the Exokernel interface is architecture dependent, dedicated to software-controlled translation lookalike buffers (TLBs). The basic communication primitive is the protected control transfer that crosses address spaces but does not transfer arguments. A lightweight RPC based on this primitive takes 10 µs on a Mips R3000 processor, while a Mach RPC needs 95 µs. Unanswered is the question of whether the right abstractions perform better and lead to better-structured and more efficient applications than Exokernel's primitives do.
- **L4.** Developed at GMD in 1995, L4 is based on the theses that efficiency and flexibility require a minimal set of general microkernel abstractions and that microkernels are processor dependent. In [16], we show that even such compatible processors as the 486 and the Pentium need different microkernel implementations (with the same API)—not only different coding but different algorithms and data structures. Like optimizing code generators, microkernels are inherently not portable, although they improve the portability of a whole system. L4 supplies three abstractions—

[2]Wiring specific pages in memory is not enough. To control second-level cache usage, DMA and management of memory areas with specific hardware-defined semantics, you need complete allocation control. The same control is needed for deallocation to enable application-specific checkpointing and swapping.

address spaces (described in the next section), threads, and IPC—implements only seven system calls, and needs only 12 Kbytes of code. Across-address-space IPC on a 486-DX50 takes 5 µs for an 8-byte argument and 18 µs for 512 bytes. The corresponding Mach numbers are 115 µs (8 bytes) and 172 µs (512 bytes). With 2 x 5 µs, the basic L4-RPC is twice as fast as a conventional Unix system call. It remains unknown whether L4's abstractions, despite being substantially more flexible than the abstractions of the first generation, are flexible and powerful enough for all types of operating systems.


B
receive ('map,' virt. addr. in B)
map
IPC
send ('map,' virt. addr. in A)
A
Figure 5. Recursively constructed address spaces

Both approaches seem to overcome the performance problem. Exokernel's and L4's communication is up to 20 times faster than that of first-generation IPC.

Some nonmicrokernel systems try to reduce communication costs by avoiding IPC. As with Chorus and Mach, Solaris and Linux support kernel-loadable modules. The Spin system [2] extends the Synthesis [20] idea and uses a kernel-integrated compiler to generate safe code inside the kernel space. Communicating with servers of this kind requires fewer address-space switches. The reduced IPC costs of second-generation microkernels might make this technique obsolete or even disqualify it, since kernel compilers impose overhead on the kernel. However, the question of which is superior—kernel-compiler technology or a pure microkernel approach—is open as long as there is no sound implementation integrating a kernel-compiler with a second-level microkernel.

More Flexibility

Performance-related constraints seem to be disappearing. The problem of first-generation microkernels was the limitation of the external-pager concept hardwiring a policy inside the kernel. This limitation was largely removed by L4's address-space concept, which provides a pure mechanism interface. Instead of offering a policy, the kernel's role is confined to offering the basic mechanisms required to implement the appropriate policies. These basic mechanisms permit implementation of various protection schemes and even of physical memory management on top of the microkernel.

The idea is to support the recursive construction of address spaces outside the kernel (see Figure 5). An initial address space represents the physical memory and is controlled by the first address-space server. At system start time, all other address spaces are empty. For construction and maintenance of further address spaces on top of the initial space, the microkernel provides three operations: grant, map, and demap.

The owner[3] of an address space can *grant* any of its pages to another space, provided the recipient agrees. The granted page is removed from the granter's address space and is included in the grantee's address space. The important restriction is that instead of physical page frames, the granter can grant only those pages already accessible to itself. The owner of an address space can also *map* any of its pages into another address space, if the recipient agrees. Afterward, the page can be accessed in both address spaces. In contrast to granting, in mapping, the page is not removed from the mapper's address space. As in the granting case, the mapper can map pages to which it already has access.

The owner of an address space can *demap* any of its pages. The demapped page remains accessible in the demapper's address space but is removed from all other address spaces that received the page directly or indirectly from the demapper. Although explicit consent of the address-space owners is not required, the operation is safe, since it is restricted to owned pages. The users of these pages already agreed to accept a potential demapping when they received the pages by mapping or granting. See the sidebar "Frequently Asked Questions on Memory Servers."

Since mapping a page requires consent between mapper and mappee, as does granting, it is implemented by IPC. In Figure 6, the mapper A sends a map message to the mappee B specifying by an appropriate receive operation that it is willing to accept a mapping and determines the virtual address of the page inside its own address space.

The address-space concept leaves memory management and paging outside the microkernel; only the grant, map, and demap operations are retained inside the kernel. Mapping and demapping are required to implement memory managers and pagers on top of the microkernel. Granting is used only in special situations to avoid double bookkeeping and address-space overflow. For a more detailed description, see [16].

In contrast to the external-pager concept, the kernel confines itself to mechanisms. Policies are left

[3]"Owner" means the thread or threads that execute inside the address space.

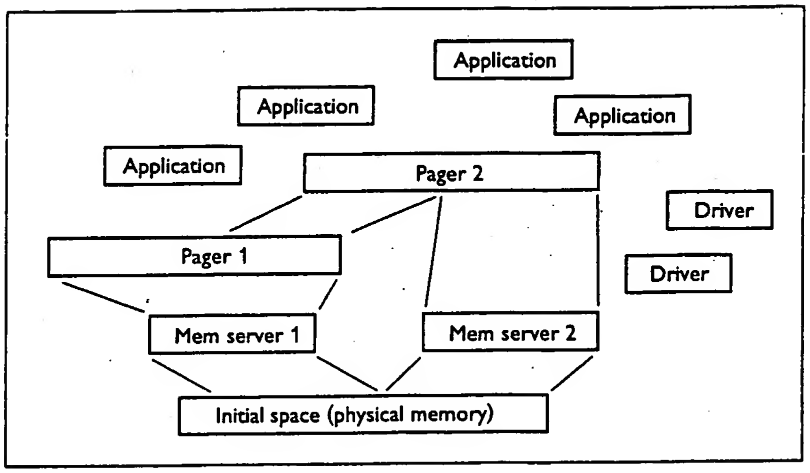


Figure 6. A maps page by IPC to B

completely to user-level servers. To illustrate, we sketch some low-level services that can be implemented by address-space servers based on the mechanisms. A server managing the initial address space is a classic main memory manager, though outside the microkernel. Memory managers can easily be stacked; the initial memory server maps or grants parts of the physical memory to memory server 1 and memory server 2. Now we have two coexisting main-memory managers.

A pager may be integrated with a memory manager or use a memory-managing server. Pagers use the microkernel's grant, map, and demap primitives. The remaining interfaces, pager-client, pager-memory server, and pager-device driver, are completely based on IPC and are defined outside the kernel. Pagers can be used to implement traditional paged virtual memory and file/database mapping, as well as unpaged resident memory for device drivers and real-time or multimedia systems. User-supplied paging strategies [5, 14] are handled at the user level and are in no way restricted by the microkernel. Stacked address spaces, like those in Grasshopper [18], and stacked file systems [13] can be realized in the same fashion.

Multimedia and other real-time applications require allocation of memory resources in a way that allows predictable execution times. For example, user-level memory managers and pagers permit fixed allocation of physical memory for specific data or for locking data in memory for a given time. Multimedia and timesharing resource allocators can coexist if the servers cooperate.

Such memory-based devices as bitmap displays are realized by a memory manager holding the screen memory in its address space.

Improving the hit rates of a secondary cache by means of page allocation or reallocation [12, 21] can be implemented through a pager that applies cache-dependent policies for allocating virtual pages in physical memory.

Remote IPC is implemented by communication servers translating local messages to external communication protocols and vice versa. The communication hardware is accessed by device drivers. If special sharing of communication buffers and user address spaces is required, the communication server also acts as a special pager for the client. In contrast to the first generation, there is no packet filter inside the microkernel.

Unix system calls are implemented by IPC. The Unix server can act as a pager for its clients and can share memory for communication with its clients. The Unix server itself is pageable or resident.

Conclusion

Although academic experiments in porting applications and operating system personalities to second-generation microkernels look promising, we have covered only the known problems of microkernels. There is no real-life practical experience with second-

Frequently Asked Questions on Memory Servers

Is a kernel without main-memory management still a microkernel or is it a submicrokernel? It is a kernel because it is mandatory to all other levels. There is no alternative kernel, although alternative memory servers may coexist. It is a microkernel because it is a direct result of applying the microkernel paradigm of making the kernel minimal. The insight that microkernels have to be much smaller than in the first generation does not justify a new "submicro" term. The underlying paradigm is the same.

Since the kernel is not usable without a memory server, why not include it in the microkernel? Two reasons:

- Operating systems can offer coexisting alternative memory servers. Examples are timesharing (paging), real-time, multimedia, and file caching techniques.
- Memory servers may be as machine (not processor) dependent as device drivers.

Specialized servers are required for controlling second-level caches and device-specific memories. If a machine has fast and slow (uncached) main-memory regions, a corresponding memory server might ensure that only fast memory is used for paging and slow memory is reserved for disk caching.

generation microkernels to draw on. Although we are optimistic, second-generation microkernel design is still research, and new problems can arise.

Most older microkernels evolved from monolithic kernels and did not achieve sufficient flexibility and performance. Although theoretically advantageous, the microkernel approach was never widely accepted in practice. However, a new generation of microkernel architectures shows promising results, and performance and flexibility have improved by an order of magnitude. Still debatable is whether Exokernel's nonabstractions, Spin's kernel compiler, L4's address-space concept, or a synthesis of these approaches is the best way forward. In each case, we expect efficient and flexible operating systems based on second-generation microkernels to be developed.

The microkernel approach was the first software architecture to be examined in detail from the performance point of view. We learned that applying the performance criterion to such a complex system is not trivial. Naive, uninterpreted measurements are sometimes misleading. Although early microkernel measurements suggested reducing the frequency of user-to-user IPC, the real problem was the structure and implementation of the kernels. To avoid such misinterpretations in the future, we should always try to understand why we get the measured results. As in physics, computer science should regard measurements as experiments used to validate or repudiate a theory.

Although steady evolution is a powerful methodology, sometimes a radically new approach is needed. Most problems of the first-generation microkernels were caused by their step-by-step development. The microkernels designed from scratch gave completely different results that could not have been extrapolated from previous experience. ◘

References

1. Bernabeu-Auban, J.M., Hutto, P.W., and Khalidi, Y.A. The architecture of the Ra kernel. Tech. Rep. GIT-ICS-87/35, Georgia Institute of Technology, Atlanta, 1988.
2. Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C. Extensibility, safety, and performance in the Spin operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)* (Copper Mountain Resort, Colo, Dec. 1995). ACM Press, 1995, pp. 267–284.
3. Bricker, A., Gien, M., Guillemont, M., Lipkis, J., Orr, D., and Rozier, M. A new look at microkernel-based Unix operating systems. Tech. Rep. CS/TR-91-7, Chorus systèmes, Paris, France, 1991.
4. Campbell, R., Islam, N., Madany, P., and Raila, D. Designing and implementing Choices: An object-oriented system in C++. *Commun. ACM 36*, 9 (Sept. 1993), 117–126.
5. Cao, P., Felten, E.W., and Li, K. Implementation and performance of application-controlled file caching. In *Proceedings of the 1st Usenix Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, Calif., Nov. 1994). ACM Press, New York, 1994, pp. 165–178.
6. Chen, J.B. and Bershad, B.N. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)* (Asheville, N.C., Dec. 1993). ACM Press, 1993, pp. 120–133.
7. Cheriton, D.R., Whitehead, G.R., and Sznyter, E.W. Binary emulation of Unix using the V kernel. In *Proceedings of the Usenix Summer Conference* (Anaheim, Calif., June 1990), pp. 73–86.
8. Condict, M., Bolinger D., McManus, E., Mitchell, D., and Lewontin, S. Microkernel modularity with integrated kernel performance. Tech. Rep., OSF Research Institute, Cambridge, Mass, 1994.
9. Engler, D., Kaashoek, M.F., and O'Toole, J. Exokernel, an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)* (Copper Mountain Resort, Colo., Dec. 1995) ACM Press, 1995, pp. 251–266.
10. Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an application program. In *Proceedings of the Usenix Summer Conference* (Anaheim, Calif., June 1990). Usenix Association, 1990, pp. 87–96.
11. Guillemont, M. The Chorus distributed operating system: Design and implementation. In *Proceedings of the ACM International Symposium on Local Computer Networks* (Firenze, Italy, Apr. 1982) ACM Press, 1982, pp. 207–223.
12. Kessler, R., and Hill, M.D. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst. 10*, 4 (Nov. 1992), 11–22.
13. Khalidi, Y.A., and Nelson, M.N. Extensible file systems in Spring. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)* (Asheville, N.C., Dec. 1993). ACM Press, New York, 1993, pp. 1–14.
14. Lee, C.H., Chen, M.C., and Chang, R.C. HiPEC: High performance external virtual memory caching. In *Proceedings of the 1st Usenix Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, Calif., Nov. 1994). ACM Press, New York, 1994, pp. 153–164.
15. Liedtke, J. A persistent system in real use—experiences of the first 13 years. In *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems (IWOOOS)* (Asheville, N.C., Dec. 1993) IEEE Computer Society Press, Washington, 1993, pp. 2–11.
16. Liedtke, J. On microkernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)* (Copper Mountain Resort, Colo., Dec. 1995). ACM Press, New York, 1995, pp. 237-250.
17. Liedtke, J., Bartling, U., Beyer, U., Heinrichs, D., Ruland, R., and Szalay, G. Two years of experience with a microkernel based operating system. *Oper. Syst. Rev. 25*, 2 (Apr. 1991), 51–62.
18. Lindstroem, A., Rosenberg, J., and Dearle, A. The grand unified theory of address spaces. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)* (Orcas Island, Wash., May 1995).
19. Mullender, A.J. The Amoeba distributed operating system: Selected papers 1984–1987. Tech. Rep. Tract. 41, CWI, Amsterdam, 1987.
20. Pu, C., Massalin, H., and Ioannidis, J. The Synthesis kernel. *Comput. Syst. 1*, 1 (Jan. 1988), 11–32.
21. Romer, T.H., Lee, D.L., Bershad, B.N., and Chen, B. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Proceedings of the 1st Usenix Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, Calif., Nov. 1994). ACM Press, New York, 1994, pp. 255–266.
22. Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating System Principles (SOSP)* (Austin, Tex., Nov. 1987). ACM Press, New York, 1987.

JOCHEN LIEDTKE is a senior researcher in the German National Research Center for Information Technology (GMD) and at IBM's T.J. Watson Research Laboratory. He can be reached at jochen.liedtke@gmd.de

09-0: Roadmap for next few weeks

- The process model (Section 2.1)
- OS structure (Section 1.5)
- MINIX structure and implementation (Section 2.5-2.6)
- Process Scheduling (Section 2.4)
- Concurrent Programming (Section 2.2-2.3)

09-1: The Process Model

- process – program in execution
- process creation in UNIX: `fork()`
- program execution in UNIX: `execl()` and friends
- multiprogramming – simulating concurrent execution on a single CPU
- scheduler – part of kernel that chooses which process to run anytime there is a choice
- quantum (or timeslice) – the amount of time a process is allowed to run before switching to another process

09-2: Processes in execution

- Picture of three processes: A, B, C
- Questions
    - Will a process always use a full quantum?
    - What is good about a short quantum?
    - What is bad about a short quantum?

09-3: States and transitions

- The states
    - ready – able to execute
    - running – actually executing
    - blocked – waiting for something, e.g., I/O, pipe data
- The transitions
    - (1) process *blocks* for, say, input
    - (2) process reaches end of quantum
    - (3) scheduler chooses a process to run
    - (4) event for which a process is waiting occurs
- Picture

09-4: Implementing processes

- Process table
    - Kernel list of processes
    - Each process represented at a C struct, usually called the *process control block* (PCB)
    - PCB fields: registers, program counter, stack pointer, process id, CPU time used, mem map, user and group info, file info
- On a single CPU system only one process can run at a time

09-5: Interrupts and processes

- Interrupt handling
    - I/O – can move process from blocked to ready
    - Clock – can move process from running to ready and another from ready to running
- Context switching
    - Each process has a stack
    - When interrupted or when blocking, save registers on stack or in process table entry
    - To resume, pop registers from stack or restore registers from process table entry

09-6: Clock interrupt handler

```
save context
decrement count /* quantum counter for *this* process */
if (count == 0) {
  /* end of slice (quantum) */
  call scheduler to choose next process to execute
  count = new_quantum /* picked by scheduler */
                      /* or just a constant */
}
restore context
```

09-7: OS structure

- So far we have only consider one type of OS structure:
    - User-level processes and a kernel
- Many ways to provide an execution environment for applications:
    - Monolithic kernels
    - Layer approaches
    - Client-server approaches
    - Virtual machines

09-8: **Monolithic kernels**

- Kernel is one large program (structured internally)
- Two modes: user mode and kernel mode
- Typical or many operating systems: Linux, Solaris, BSD, Windows 98
- Processes request services from kernel using system calls
- Pro: simple structure, fast
- Con: internal design is complex, bad module or driver can crash kernel

09-9: **Layered approach**

- Kernel is a series of software layer
- Each layer is built on top of a lower layer
- Abstraction helps hide system complexity
- Found in older operating systems: THE, MULTICS
- Some layering also found in OS/2 and Windows NT
- Pro: robust, each layer is responsible for well defined tasks
- Con: poor performance

09-10: **Client-server approaches**

- Kernel is a minimal core that provides process, memory management, and inter-process communication
- Higher-level services such as device drivers, file systems, and network protocols are implemented as *servers* that run as user-level processes
- Also called the *microkernel approach*
- Examples: MINIX, Amoeba, Mach, MkLinux, GNU Hurd, Windows NT
- Pro: good for organization and protection
- Con: bad for performance

09-11: **Virtual machines**

- Allow multiple operating systems to run on a single machine simultaneously
- Each *virtual machine* gives the guest OS the illusion of running on the real hardware
- Kernel is a *virtual machine monitor*
- All I/O and hardware instructions must be trapped
- Examples: Early IBM machines (VM/370), VMware for Linux and Windows

09-12: **MINIX structure**

- MINIX uses the client-server approach
- The lowest level provides process management and message passing
- Everything else runs as a process (e.g., device drivers, memory management, the file system, the networking, and user processes)
- All processes communicate using *message passing*

09-13: **MINIX structure continued 1**

Level	Kind	Specifi cs
4	user processes	init, user processes, daemons
3	server processes	memory manager and fi le system
2	I/O tasks	disk, tty clock, system, etc.
1		processes and message passing

- MINIX is conceptually divided into four levels
- The *MINIX kernel* consists of levels 1 and 2

09-14: **MINIX structure continued 2**

- Process management: handles interrupts, context switching, low-level memory management, message passing
- Server processes: provide the system call interface to the user processes
- One init process: the first process that starts all other processes
- Addressability – separate programs for: kernel, MM (memory management), FF (file system server), init (first process)
- However, all of these are rolled into a single file and loaded together at boot time

09-15: **MINIX message passing**

- All processes communication using messages
- Messages are *fixed size* (a fixed number of bytes)
- Three primitives
  - `send(dest, &message)`
  - `receive(source, &message)`
  - `send_rec(src_dst, &message)`
- User processes *cannot* use message passing to communicate with other user processes, they can only send messages to the server processes
- Servers can communicate with servers and tasks
- Tasks can communicate with tasks

09-16: MINIX message passing cont

- Three primitives
  - `send(dest, &message)`
  - `receive(source, &message)`
  - `send_rec(src_dst, &message)`
- `dest`, `source`, and `src_dst` are process indexes; source can be ANY.
- blocking send and receive (*rendezvous*)

09-17: MINIX scheduling

- MINIX uses a round robin scheduler with priorities

Level	Process type
2	user processes
1	server processes
0	I/O tasks

- Lower-level processes scheduled first
- RR is used in each level
- Tasks and servers are never preempted, only user processes

09-18: MINIX source code

- Source in back of the book
- Will be hard to understand at first
- Need to read the book and source at the same time
- Use bookmarkers to identify important sections
- The book source is slightly different from the real source
  - The real source is version 2.0.3 (the book is 2.0.0)
  - The real source includes support for multiple platforms

09-19: MINIX source: key points

- Header files
  - Message layout
  - Process table
  - Ready lists
- Code
  - System startup: main called from MINIX, initializes tasks and servers
  - Interrupt handling
  - rendezvous (sleep and wakeup)
  - Scheduling

text from: http://www.cs.vu.nl/~ast/brown/

Some Notes on the "Who wrote Linux" Kerfuffle, Release 1.5

Original comment Follow up Code comparison Rebuttal

Background
The history of UNIX and its various children and grandchildren has been in the news recently as a result of a book from the Alexis de Tocqueville Institution. Since I was involved in part of this history, I feel I have an obligation to set the record straight and correct some extremely serious errors. But first some background information.

Ken Brown, President of the Alexis de Tocqueville Institution, contacted me in early March. He said he was writing a book on the history of UNIX and would like to interview me. Since I have written 15 books and have been involved in the history of UNIX in several ways, I said I was willing to help out. I have been interviewed by many people for many reasons over the years, and have been on Dutch and US TV and radio and in various newspapers and magazines, so I didn't think too much about it.

Brown flew over to Amsterdam to interview me on 23 March 2004. Apparently I was the only reason for his coming to Europe. The interview got off to a shaky start, roughly paraphrased as follows:
AST: "What's the Alexis de Tocqueville Institution?"
KB: We do public policy work
AST: A think tank, like the Rand Corporation?
KB: Sort of
AST: What does it do?
KB: Issue reports and books
AST: Who funds it?
KB: We have multiple funding sources
AST: Is SCO one of them? Is this about the SCO lawsuit?
KB: We have multiple funding sources
AST: Is Microsoft one of them?
KB: We have multiple funding sources

He was extremely evasive about why he was there and who was funding him. He just kept saying he was just writing a book about the history of UNIX. I asked him what he thought of Peter Salus' book, A Quarter Century of UNIX. He'd never heard of it! I mean, if you are writing a book on the history of UNIX and flying 3000 miles to interview some guy about the subject, wouldn't it make sense to at least go to amazon.com and type "history unix" in the search box, in which case Salus' book is the first hit? For $28 (and free shipping if you play your cards right) you could learn an awful lot about the material and not get any jet lag. As I soon learned, Brown is not the sharpest knife in the drawer, but I was already suspicious. As a long-time author, I know it makes sense to at least be aware of what the competition is. He didn't bother.

UNIX and Me

I didn't think it odd that Brown would want to interview me about the history of UNIX. There are worse people to ask. In the late 1970s and early 1980s, I spent several summers in the UNIX group (Dept. 1127) at Bell Labs. I knew Ken Thompson, Dennis Ritchie, and the rest of the people involved in the development of UNIX. I have stayed at Rob Pike's house and Al Aho's house for extended periods of time. Dennis Ritchie, Steve Johnson, and Peter Weinberger, among others have stayed at my house in Amsterdam. Three of my Ph.D. students have worked in the UNIX group at Bell Labs and one of them is a permanent staff member now.

Oddly enough, when I was at Bell Labs, my interest was not operating systems, although I had written one and published a paper about it (see "Software - Practice & Experience," vol. 2, pp. 109-119, 1973). My interest then was compilers, since I was the chief designer of the Amsterdam Compiler Kit (see Commun. of the ACM, vol. 26, pp. 654-660, Sept. 1983.). I spent some time there discussing compilers with Steve Johnson, networking with Greg Chesson, writing tools with Lorinda Cherry, and book authoring with Brian Kernighan, among many others. I also became friends with the other "foreigner," there, Bjarne Stroustrup, who would later go on to design and implement C++.

In short, although I had nothing to do with the development of the original UNIX, I knew all the people involved and much of the history quite well. Furthermore, my contact with the UNIX group at Bell Labs was not a secret; I even thanked them all for having me as a summer visitor in the preface to the first edition of my book Computer Networks. Amazingly, Brown knew nothing about any of this. He didn't do his homework before embarking on his little project

MINIX and Me

Years later, I was teaching a course on operating systems and using John Lions' book on UNIX Version 6. When AT&T decided to forbid the teaching of the UNIX internals, I decided to write my own version of UNIX, free of all AT&T code and restrictions, so I could teach from it. My inspiration was not my time at Bell Labs, although the knowledge that one person could write a UNIX-like operating system (Ken Thompson wrote UNICS on a PDP-7) told me it could be done. My real inspiration was an off-hand remark by Butler Lampson in an operating systems course I took from him when I was a Ph.D. student at Berkeley. Lampson had just finished describing the pioneering CTSS operating system and said, in his inimitable way: "Is there anybody here who couldn't write CTSS in a month?" Nobody raised his hand. I concluded that you'd have to be real dumb not to be able to write an operating system in a month. The paper cited above is an operating system I wrote at Berkeley with the help of Bill Benson. It took a lot more than a month, but I am not as smart as Butler. Nobody is.

I set out to write a minimal UNIX clone, MINIX, and did it alone. The code was 100% free of AT&T's intellectual property. The full source code was published in 1987 as the appendix to a book, Operating Systems: Design and Implementation, which later went into a second edition co-authored with Al Woodhull. MINIX 2.0 was even POSIX-conformant. Both editions contained hundreds of pages of text describing the code in great detail. A box of 10 floppy disks containing all the binaries and source code was available separately from Prentice Hall for $69.

While this was not free software in the sense of "free beer" it was free software in the sense of "free speech" since all the source code was available for only slightly more than the manufacturing cost. But even "free speech" is not completely "free"--think about slander, yelling "fire" in a crowded theater, etc. Also Remember (if you are old enough) that by 1987, a university educational license for UNIX cost $300, a commercial license for a university cost $28,000, and a commercial license for a company cost a lot more. For the first time, MINIX brought the cost of "UNIX-like" source code down to something a student could afford. Prentice Hall wasn't really interested in selling software. They were interested in selling books, so there was a fairly liberal policy on copying MINIX, but if a company wanted to sell it to make big bucks, PH wanted a royalty. Hence the PH lawyers equipped MINIX with a lot of boilerplate, but there was never any intention of really enforcing this against universities or students. Using the Internet for distributing that much code was not feasible in 1987, even for people with a high-speed (i.e., 1200 bps) modem. When distribution via the Internet became feasible, I convinced Prentice Hall to drop its (extremely modest) commercial ambitions and they gave me permission to put the source on my website for free downloading, where it still is.

Within a couple of months of its release, MINIX became something of a cult item, with its own USENET newsgroup, comp.os.minix, with 40,000 subscribers. Many people added new utility programs and improved the kernel in numerous ways, but the original kernel was just the work of one person--me. Many people started pestering me about improving it. In addition to the many messages in the USENET newsgroup, I was getting 200 e-mails a day (at a time when only the chosen few had e-mail at all) saying things like: "I need pseudoterminals and I need them by Friday." My answer was generally quick and to the point: "No."

The reason for my frequent "no" was that everyone was trying to turn MINIX into a production-quality UNIX system and I didn't want it to get so complicated that it would become useless for my purpose, namely, teaching it to students. I also expected that the niche for a free production-quality UNIX system would be filled by either GNU or Berkeley UNIX shortly, so I wasn't really aiming at that. As it turned out, the GNU OS sort of went nowhere (although many UNIX utilities were written) and Berkeley UNIX got tied up in a lawsuit when its designers formed a company, BSDI, to sell it and they chose 1-800-ITS UNIX as their phone number. AT&T felt this constituted copyright infringement and sued them. It took a couple of years for this to get resolved. This delay in getting free BSD out there gave Linux the breathing space it needed to catch on. If it hadn't been for the lawsuit, undoubtedly BSD would have filled the niche for a powerful, free UNIX clone as it was already a stable, mature system with a large following.

Ken Brown and Me

Now Ken Brown shows up and begins asking questions. I quickly determined that he didn't know a thing about the history of UNIX, had never heard of the Salus book, and knew nothing about BSD and the AT&T lawsuit. I started to tell him the history, but he stopped me and said he was more interested in the legal aspects. I said: "Oh you mean about Dennis Ritchie's patent number 4135240 on the setuid bit?" Then I added:"That's not a problem. Bell Labs dedicated the patent." That's when I discovered that (1) he had never heard of the patent, (2) did not know what it meant to dedicate a patent (i.e., put it in the public domain), and (3) really did not know a thing about intellectual property law. He was confused about patents, copyrights, and trademarks. Gratuitously, I asked if he was a lawyer, but it was obvious he was not and he admitted it. At this point I was still thinking he might be a spy from SCO, but if he was, SCO was not getting its money's worth.

He wanted to go on about the ownership issue, but he was also trying to avoid telling me what his real purpose was, so he didn't phrase his questions very well. Finally he asked me if I thought Linus wrote Linux. I said that to the best of my knowledge, Linus wrote the whole kernel himself, but after it was released, other people began improving the kernel, which was very primitive initially, and adding new software to the system--essentially the same development model as MINIX. Then he began to focus on this, with questions like: "Didn't he steal pieces of MINIX without permission." I told him that MINIX had clearly had a huge influence on Linux in many ways, from the layout of the file system to the names in the source tree, but I didn't think Linus had used any of my code. Linus also used MINIX as his development platform initially, but there was nothing wrong with that. He asked if I objected to that and I said no, I didn't, people were free to use it as they wished for noncommercial purposes. Later MINIX was released under the Berkeley license, which freed it up for all purposes. It is still in surprisingly wide use, both for education and in the Third World, where millions of people are happy as a clam to have an old castoff 1-MB 386, on which MINIX runs just fine. The MINIX home page cited above still gets more than 1000 hits a week.

Finally, Brown began to focus sharply. He kept asking, in different forms, how one person could write an operating system all by himself. He simply didn't believe that was possible. So I had to give him more history, sigh. To start with, Ken Thompson wrote UNICS for the PDP-7 all by himself. When it was later moved to the PDP-11 and rewritten in C, Dennis Ritchie joined the team, but primarily focused on designing the C language, writing the C compiler, and writing the I/O system and device drivers. Ken wrote nearly all of the kernel himself.

In 1983, a now-defunct company named the Mark Williams company produced and sold a very good UNIX clone called Coherent. Most of the work was done by three ex-students from the University of Waterloo: Dave Conroy, Randall Howard, and Johann

George. It took them two years. But they produced not only the kernel, but the C compiler, shell, and ALL the UNIX utilities. This is far more work than just making a kernel. It is likely that the kernel took less than a man-year.

In 1983, Ric Holt published a book, now out of print, on the TUNIS system, a UNIX-like system. This was certainly a rewrite since TUNIS was written in a completely new language, concurrent Euclid.

Then Doug Comer wrote XINU. While also not a UNIX clone, it was a comparable system.

In addition, Gary Kildall wrote CP/M by himself and Tim Paterson wrote MS-DOS. While these systems from the early 1980s were not even close to being UNIX-clones, they were substantial and popular operating systems written by individuals.

By the time Linus started, five people or small teams had independently implemented the UNIX kernel or something approximating it, namely, Thompson, Coherent, Holt, Comer, and me. All of this was perfectly legal and nobody stole anything. Given this history, it is pretty hard to make a case that one person can't implement a system of the complexity of Linux, whose original size was about the same as V1.0 of MINIX.

Of course it is always true in science that people build upon the work of their predecessors. Even Ken Thompson wasn't the first. Before writing UNIX, Ken had worked on the MIT MULTICS (MULTiplexed Information and Computing Service) system. In fact, the original name of UNIX was UNICS, a joke made by Brian Kernighan standing for the UNIplexed Information and Computing Service, since the PDP-7 version could support only one user--Ken. After too many bad puns about EUNUCHS being a castrated MULTICS, the name was changed to UNIX. But even MULTICS wasn't first. Before it was the above-mentioned CTSS, designed by the same team at MIT.

Thus, of course, Linus didn't sit down in a vacuum and suddenly type in the Linux source code. He had my book, was running MINIX, and undoubtedly knew the history (since it is in my book). But the code was his. The proof of this is that he messed the design up. MINIX is a nice, modular microkernel system, with the memory manager and file system running as user-space processes. This makes the system cleaner and more reliable than a big monolithic kernel and easier to debug and maintain, at a small price in performance, although even on a 4.77 MHz 8088 it booted in maybe 5 seconds (vs. a minute for Windows on hardware 500 times faster). An example of commercially successful microkernel is QNX. Instead of writing a new file system and a new memory manager, which would have been easy, Linus rewrote the whole thing as a big monolithic kernel, complete with inline assembly code :-(. The first version of Linux was like a time machine. It went back to a system worse than what he already had on his desk. Of course, he was just a kid and didn't know better (although if he had paid better attention in class he should have), but producing a system that was fundamentally different from the base he started with seems pretty good proof that it was a redesign. I don't think he could have copied UNIX because he didn't have access to the UNIX source code, except maybe John

Lions' book, which is about an earlier version of UNIX that does not resemble Linux so much.

My conclusion is that Ken Brown doesn't have a clue what he is talking about. I also have grave questions about his methodology. After he talked to me, he prowled the university halls buttonholing random students and asking them questions. Not exactly primary sources.

The six people I know of who (re)wrote UNIX all did it independently and nobody stole anything from anyone. Brown's remark that people have tried and failed for 30 years to build UNIX-like systems is patent nonsense. Six different people did it independently of one another. In science it is considered important to credit people for their ideas, and I think Linus has done this far less than he should have. Ken and Dennis are the real heros here. But Linus' sloppiness about attribution is no reason to assert that Linus didn't write Linux. He didn't write CTSS and he didn't write MULTICS and didn't write UNIX and he didn't write MINIX, but he did write Linux. I think Brown owes a number of us an apology.

Linus and Me
Some of you may find it odd that I am defending Linus here. After all, he and I had a fairly public "debate" some years back. My primary concern here is trying to get the truth out and not blame everything on some teenage girl from the back hills of West Virginia. Also, Linus and I are not "enemies" or anything like that. I met him once and he seemed like a nice friendly, smart guy. My only regret is that he didn't develop Linux based on the microkernel technology of MINIX. With all the security problems Windows has now, it is increasingly obvious to everyone that tiny microkernels, like that of MINIX, are a better base for operating systems than huge monolithic systems. Linux has been the victim of fewer attacks than Windows because (1) it actually is more secure, but also (2) most attackers think hitting Windows offers a bigger bang for the buck so Windows simply gets attacked more. As I did 20 years ago, I still fervently believe that the only way to make software secure, reliable, and fast is to make it small. Fight Features.

If you have made it this far, thank you for your time. Permission is hereby granted to mirror this web page provided that the original, unmodified version is used.

Andy Tanenbaum, 20 May 2004

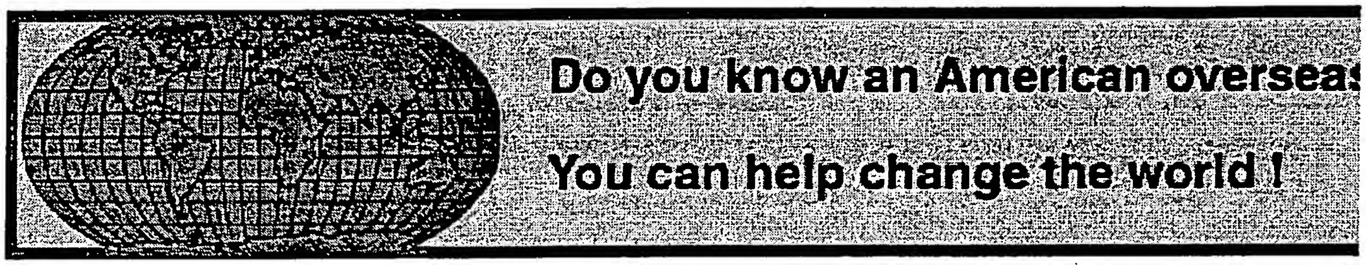
Followup statement on Ken Brown's motivation written 21 May 2004

Back to my home page

Your browser does not fully support CSS. This may result in visual artifacts.

faculty of sciences
department of computer science

vrije Universiteit amsterdam


Do you know an American overseas
You can help change the world !

Some Notes on the "Who wrote Linux" Kerfuffle, Releas

Original comment Follow up Code comparison Rebuttal

Background

The history of UNIX and its various children and grandchildren has been in the news recently as a result of
Alexis de Tocqueville Institution. Since I was involved in part of this history, I feel I have an obligation to se
straight and correct some extremely serious errors. But first some background information.

Ken Brown, President of the Alexis de Tocqueville Institution, contacted me in early March. He said he was
on the history of UNIX and would like to interview me. Since I have written 15 books and have been involv
of UNIX in several ways, I said I was willing to help out. I have been interviewed by many people for many
years, and have been on Dutch and US TV and radio and in various newspapers and magazines, so I didn't
about it.

Brown flew over to Amsterdam to interview me on 23 March 2004. Apparently I was the only reason for his
Europe. The interview got off to a shaky start, roughly paraphrased as follows:
AST: "What's the Alexis de Tocqueville Institution?"
KB: We do public policy work
AST: A think tank, like the Rand Corporation?
KB: Sort of
AST: What does it do?
KB: Issue reports and books
AST: Who funds it?
KB: We have multiple funding sources
AST: Is SCO one of them? Is this about the SCO lawsuit?
KB: We have multiple funding sources
AST: Is Microsoft one of them?
KB: We have multiple funding sources

He was extremely evasive about why he was there and who was funding him. He just kept saying he was jı
book about the history of UNIX. I asked him what he thought of Peter Salus' book, A Quarter Century of UN
heard of it! I mean, if you are writing a book on the history of UNIX and flying 3000 miles to interview som
subject, wouldn't it make sense to at least go to amazon.com and type "history unix" in the search box, in
Salus' book is the first hit? For $28 (and free shipping if you play your cards right) you could learn an awful
material and not get any jet lag. As I soon learned, Brown is not the sharpest knife in the drawer, but I was
suspicious. As a long-time author, I know it makes sense to at least be aware of what the competition is. H

UNIX and Me

I didn't think it odd that Brown would want to interview me about the history of UNIX. There are worse peo
late 1970s and early 1980s, I spent several summers in the UNIX group (Dept. 1127) at Bell Labs. I knew I
Dennis Ritchie, and the rest of the people involved in the development of UNIX. I have stayed at Rob Pike's
Aho's house for extended periods of time. Dennis Ritchie, Steve Johnson, and Peter Weinberger, among oth
at my house in Amsterdam. Three of my Ph.D. students have worked in the UNIX group at Bell Labs and or
permanent staff member now.

Oddly enough, when I was at Bell Labs, my interest was not operating systems, although I had written one
paper about it (see "Software - Practice & Experience," vol. 2, pp. 109-119, 1973). My interest then was co
was the chief designer of the Amsterdam Compiler Kit (see *Commun. of the ACM*, vol. 26, pp. 654-660, Sep
spent some time there discussing compilers with Steve Johnson, networking with Greg Chesson, writing too
Cherry, and book authoring with Brian Kernighan, among many others. I also became friends with the othe
there, Bjarne Stroustrup, who would later go on to design and implement C++.

In short, although I had nothing to do with the development of the original UNIX, I knew all the people invo
of the history quite well. Furthermore, my contact with the UNIX group at Bell Labs was not a secret; I eve
all for having me as a summer visitor in the preface to the first edition of my book *Computer Networks*. Am
knew nothing about any of this. He didn't do his homework before embarking on his little project

MINIX and Me

Years later, I was teaching a course on operating systems and using John Lions' book on UNIX Version 6. W
decided to forbid the teaching of the UNIX internals, I decided to write my own version of UNIX, free of all
restrictions, so I could teach from it. My inspiration was *not* my time at Bell Labs, although the knowledge t
could write a UNIX-like operating system (Ken Thompson wrote UNICS on a PDP-7) told me it could be don
inspiration was an off-hand remark by Butler Lampson in an operating systems course I took from him whe
student at Berkeley. Lampson had just finished describing the pioneering CTSS operating system and said,
way: "Is there anybody here who couldn't write CTSS in a month?" Nobody raised his hand. I concluded th
be real dumb not to be able to write an operating system in a month. The paper cited above is an operatin
at Berkeley with the help of Bill Benson. It took a lot more than a month, but I am not as smart as Butler. I

I set out to write a minimal UNIX clone, MINIX, and did it alone. The code was 100% free of AT&T's intellec
The full source code was published in 1987 as the appendix to a book, Operating Systems: Design and Imp
which later went into a second edition co-authored with Al Woodhull. MINIX 2.0 was even POSIX-conforma
contained hundreds of pages of text describing the code in great detail. A box of 10 floppy disks containing
and source code was available separately from Prentice Hall for $69.

While this was not free software in the sense of "free beer" it was free software in the sense of "free speech
source code was available for only slightly more than the manufacturing cost. But even "free speech" is not
"free"--think about slander, yelling "fire" in a crowded theater, etc. Also Remember (if you are old enough)
university educational license for UNIX cost $300, a commercial license for a university cost $28,000, and a
license for a company cost a lot more. For the first time, MINIX brought the cost of "UNIX-like" source code
something a student could afford. Prentice Hall wasn't really interested in selling software. They were inter

books, so there was a fairly liberal policy on copying MINIX, but if a company wanted to sell it to make big
wanted a royalty. Hence the PH lawyers equipped MINIX with a lot of boilerplate, but there was never any i
enforcing this against universities or students. Using the Internet for distributing that much code was not fe
even for people with a high-speed (i.e., 1200 bps) modem. When distribution via the Internet became feasi
Prentice Hall to drop its (extremely modest) commercial ambitions and they gave me permission to put the
website for free downloading, where it still is.

Within a couple of months of its release, MINIX became something of a cult item, with its own USENET new
comp.os.minix, with 40,000 subscribers. Many people added new utility programs and improved the kernel
ways, but the original kernel was just the work of one person--me. Many people started pestering me abou
addition to the many messages in the USENET newsgroup, I was getting 200 e-mails a day (at a time wher
few had e-mail at all) saying things like: "I need pseudoterminals and I need them by Friday." My answer w
quick and to the point: "No."

The reason for my frequent "no" was that everyone was trying to turn MINIX into a production-quality UNI)
didn't want it to get so complicated that it would become useless for my purpose, namely, teaching it to stu
expected that the niche for a free production-quality UNIX system would be filled by either GNU or Berkeley
so I wasn't really aiming at that. As it turned out, the GNU OS sort of went nowhere (although many UNIX
written) and Berkeley UNIX got tied up in a lawsuit when its designers formed a company, BSDI, to sell it a
800-ITS UNIX as their phone number. AT&T felt this constituted copyright infringement and sued them. It t
years for this to get resolved. This delay in getting free BSD out there gave Linux the breathing space it ne
If it hadn't been for the lawsuit, undoubtedly BSD would have filled the niche for a powerful, free UNIX clon
already a stable, mature system with a large following.

Ken Brown and Me

Now Ken Brown shows up and begins asking questions. I quickly determined that he didn't know a thing ab
of UNIX, had never heard of the Salus book, and knew nothing about BSD and the AT&T lawsuit. I started t
history, but he stopped me and said he was more interested in the legal aspects. I said: "Oh you mean abo
Ritchie's patent number 4135240 on the setuid bit?" Then I added:"That's not a problem. Bell Labs dedicat
That's when I discovered that (1) he had never heard of the patent, (2) did not know what it meant to dedi
(i.e., put it in the public domain), and (3) really did not know a thing about intellectual property law. He wa
about patents, copyrights, and trademarks. Gratuitously, I asked if he was a lawyer, but it was obvious he
admitted it. At this point I was still thinking he might be a spy from SCO, but if he was, SCO was not gettin
worth.

He wanted to go on about the ownership issue, but he was also trying to avoid telling me what his real pur
didn't phrase his questions very well. Finally he asked me if I thought Linus wrote Linux. I said that to the b
knowledge, Linus wrote the whole kernel himself, but after it was released, other people began improving t
was very primitive initially, and adding new software to the system--essentially the same development mod
Then he began to focus on this, with questions like: "Didn't he steal pieces of MINIX without permission." I
MINIX had clearly had a huge influence on Linux in many ways, from the layout of the file system to the na
source tree, but I didn't think Linus had used any of my code. Linus also used MINIX as his development pl
but there was nothing wrong with that. He asked if I objected to that and I said no, I didn't, people were fr
they wished for noncommercial purposes. Later MINIX was released under the Berkeley license, which free
purposes. It is still in surprisingly wide use, both for education and in the Third World, where millions of pe
as a clam to have an old castoff 1-MB 386, on which MINIX runs just fine. The MINIX home page cited abo
than 1000 hits a week.

Finally, Brown began to focus sharply. He kept asking, in different forms, how one person could write an op
all by himself. He simply didn't believe that was possible. So I had to give him more history, sigh. To start
Thompson wrote UNICS for the PDP-7 all by himself. When it was later moved to the PDP-11 and rewritten
Ritchie joined the team, but primarily focused on designing the C language, writing the C compiler, and wri
system and device drivers. Ken wrote nearly all of the kernel himself.

In 1983, a now-defunct company named the Mark Williams company produced and sold a very good UNIX (Coherent. Most of the work was done by three ex-students from the University of Waterloo: Dave Conroy, I and Johann George. It took them two years. But they produced not only the kernel, but the C compiler, she UNIX utilities. This is far more work than just making a kernel. It is likely that the kernel took less than a m

In 1983, Ric Holt published a book, now out of print, on the TUNIS system, a UNIX-like system. This was ce since TUNIS was written in a completely new language, concurrent Euclid.

Then Doug Comer wrote XINU. While also not a UNIX clone, it was a comparable system.

In addition, Gary Kildall wrote CP/M by himself and Tim Paterson wrote MS-DOS. While these systems from were not even close to being UNIX-clones, they were substantial and popular operating systems written by

By the time Linus started, five people or small teams had independently implemented the UNIX kernel or sc approximating it, namely, Thompson, Coherent, Holt, Comer, and me. All of this was perfectly legal and no anything. Given this history, it is pretty hard to make a case that one person can't implement a system of t Linux, whose original size was about the same as V1.0 of MINIX.

Of course it is always true in science that people build upon the work of their predecessors. Even Ken Thom first. Before writing UNIX, Ken had worked on the MIT MULTICS (MULTiplexed Information and Computing : In fact, the original name of UNIX was UNICS, a joke made by Brian Kernighan standing for the UNIplexed Computing Service, since the PDP-7 version could support only one user--Ken. After too many bad puns ab being a castrated MULTICS, the name was changed to UNIX. But even MULTICS wasn't first. Before it was t mentioned CTSS, designed by the same team at MIT.

Thus, of course, Linus didn't sit down in a vacuum and suddenly type in the Linux source code. He had my running MINIX, and undoubtedly knew the history (since it is in my book). But the code was his. The proof messed the design up. MINIX is a nice, modular microkernel system, with the memory manager and file sy: user-space processes. This makes the system cleaner and more reliable than a big monolithic kernel and ea and maintain, at a small price in performance, although even on a 4.77 MHz 8088 it booted in maybe 5 sec minute for Windows on hardware 500 times faster). An example of commercially successful microkernel is (writing a new file system and a new memory manager, which would have been easy, Linus rewrote the who monolithic kernel, complete with inline assembly code :-(. The first version of Linux was like a time machir to a system worse than what he already had on his desk. Of course, he was just a kid and didn't know bett had paid better attention in class he should have), but producing a system that was fundamentally differen he started with seems pretty good proof that it was a redesign. I don't think he could have copied UNIX bec have access to the UNIX source code, except maybe John Lions' book, which is about an earlier version of U not resemble Linux so much.

My conclusion is that Ken Brown doesn't have a clue what he is talking about. I also have grave questions a methodology. After he talked to me, he prowled the university halls buttonholing random students and aski questions. Not exactly primary sources.

The six people I know of who (re)wrote UNIX all did it independently and nobody stole anything from anyor remark that people have tried and failed for 30 years to build UNIX-like systems is patent nonsense. Six dif it independently of one another. In science it is considered important to credit people for their ideas, and I done this far less than he should have. Ken and Dennis are the real heros here. But Linus' sloppiness about reason to assert that Linus didn't write Linux. He didn't write CTSS and he didn't write MULTICS and didn't he didn't write MINIX, but he did write Linux. I think Brown owes a number of us an apology.

Linus and Me

Some of you may find it odd that I am defending Linus here. After all, he and I had a fairly public "debate" back. My primary concern here is trying to get the truth out and not blame everything on some teenage gir

hills of West Virginia. Also, Linus and I are not "enemies" or anything like that. I met him once and he seen
friendly, smart guy. My only regret is that he didn't develop Linux based on the microkernel technology of M
the security problems Windows has now, it is increasingly obvious to everyone that tiny microkernels, like t
are a better base for operating systems than huge monolithic systems. Linux has been the victim of fewer a
Windows because (1) it actually is more secure, but also (2) most attackers think hitting Windows offers a b
the buck so Windows simply gets attacked more. As I did 20 years ago, I still fervently believe that the only
software secure, reliable, and fast is to make it small. Fight Features.

If you have made it this far, thank you for your time. Permission is hereby granted to mirror this web page
the original, unmodified version is used.

Andy Tanenbaum, 20 May 2004

Followup statement on Ken Brown's motivation written 21 May 2004

Back to my home page

OS kernels :

a little overview and comparison

Preface

There are different types of operating system models around. Some are used in existing commercial/freeware operating systems, and others are being invented at universities in development projects. All OS types have their strong sides and their weaknesses, making them suited for different types of hardware or purposes. Off course, computers have changed a lot, so kernels have changed too. Older OSs are still based on the low-performant hardware of the sixties and seventies, but do deliver stability, while newer OS need teh power of the modern computers and still have to prove themselves.

Talking about the weak and strong sides of OSs is difficult, because most OSs are targeted to a specific group of users or applications or are used on a specific base of computers. There are OSs that claim to be general-purpose, but the first OSs that can handle all tasks as efficiently as one should like, has still to be written.

First, some words about the meaning of "kernel". Operating Systems can be written so that most services are moved outside the OS core and implemented as processes.This OS core then becomes a lot smaller, and we call it a kernel. When this kernel only provides the basic services, such as basic memory management ant multithreading, it is called a microkernel or even nanokernel for the super-small ones. To stress the difference between the

Unix-type of OS, the Unix-like core is called a monolithic kernel. A monolithic kernel provides full process management, device drivers,file systems, network access etc. I will here use the word kernel in the broad sense, meaning the part of the OS supervising the machine.

Most kernels offer 2 basic incapsulations of programs. The terms used to describe these differ between the OSs, but I will use these:

- processes : a process has a certain (protected) memory area for its own It has a status that can be :

running, waiting for some event etc

- threads : a thread is part of a process, it incapsulated an execution flow. Each thread has its own set of registers (thus, it's own virtual cpu) , but all threads in a process share the same memory space.

This leads us to the second important feature in operating system kernels: memory spaces. Each process has it's own protected (whatever this means in different OSs) memory space it runs in. It can share parts of this memory with other processes. Some OSs use a shared memory space for all processes. This means less or no protection however, like in DOS.

When it comes to existing "modern" and wide-spread operating systems, only monolithic kernels and microkernels are used. There are however, different types of microkernel architectures. Off course, lots of large mainframes in the world still run the good old VMS and other old operating systems. Unix, the OS with a monolithical kernel, was based on the ideas of the first successful OS: Multics.

[top]

The OS without kernel

Not all operating systems have a "kernel" wich is protected from user programs and wich manages the hardware and the user programs. Some operating systems, the early ones, just provided some interface to the hardware programs could run on, but didn't protect themselves from these programs or didn't offer protect the programs from each other. Thus, the user could do with the machine what he wanted to do,access the hardware directly. The strong side of this "architecture" is the speed of the system, but it can off course only be used as a personal system for 1 user.It's not stable at all when running several programs at once (if this is, at all, possible).

[top]

OS with ring structure, Multics

Then people found out "protection"... The OS was divided in several rings with different privileges. The parts of the OS that needed to access the hardware and provided the basic metafores of processes,memory and devices, run in ring0, some system tasks run in ring 1 etc... The normal user processes run in the rign with the lowest privileges. This means a process running in a certain ring cannot harm the processes in a ring with more privilege. Multics was the OS that brought this ideas to us, and formed the base for all later operating systems up to now. This architecture offers off course a lot more stability and security than the earlier architectures, and is able to provide multitasking and multi-user facilities. It can however only implemented on a system if the hardware (mainly the cpu and the mmu) provides a kind of protection facility.

[top]

OS with a kernel

This architecture evolved to an OS design with two rings: one ring running in system mode, and a ring running in user mode. The kernel has full control of the hardware and provides abstractions for the processes running in user mode. A process running in user mode cannot access the hardware, and must use the abstractions provided by the kernel. It can call certain services of the kernel by making "system calls" or kernel calls. The kernel only offers the basic services. All others are provided by programs running in user mode. This is generally the whole interface.

[top]

Monolithic kernels

This type of kernel can easily be described as "the big soup". The older monolithic kernels were written as a mixture of everything the OS needed, without much of an organization.The monolithic kernel offers everything the OS needs : processes, memory management, multiprogramming, interprocess communication (IPC), device access, file systems, network protocols and whatever the OS should implement.
Newer monolithic kernels have a modular design, which offers run-time adding and removal of services. The whole kernel runs in "kernel mode", a processor mode in which the software has full control over teh machine. The processes running on top of the kernel run in "user mode", in which programs have only access to the kernel services.
The monolithical kernels (mainly Unix) were build for systems that don't have to turned on and off every day or hour and for systems to which no devices are added during their lifetime or hardware is changed. The manufacturer tailors the OS to the machine and then sells the machine with the OS installed for once and always.
This leads to a very stable system, but it is not very suited for PCs, where devices are added or removed and that reboot every day. It is a fact, however that it's just these OSs that now even offer hot-swapping of devices etc. e.g., the new Solaris can even hot-swap cpu boards ! The main strong sides of monolithic systems is that they are extremely stable and that they achieve a very high speed.
Also, they have been in the world for so long (Unix was developped in '69, first as a single-user OS for the minicomputer)

[top]

Microkernels

Microkernel designs put a lot of OS services in seperate processes, that can be started or stopped at runtime.This makes the kernel a lot smaller and offers a far greater flexibility.
File systems, device drivers ,process management and even part of the memory management can be put in processes running on top of the microkernel. This architecture is actually a client-server (what a buzz-word :-)) model: processes (clients) can call OS services by sending requests through IPC to server processes. eg,. a process that wants to read from a certain file send a request to the file system process.The central processes that provide the process management, file system etc are frequently called the servers. Microkernels are often also highly multithreaded, putting every different service in a different thread, offering greater speed and stability. The main difficulty of microkernels is then to make the IPC as fast as possible. This was a design problem in early microkernel design, because IPC, while being intended to be the power of the architecture, often proved to be the bottleneck. Now, however, microkernels do offer fast IPC.
When talking about microkernels, one must really clearly make a difference between the first and the second generation. The first-generation microkernels, like Mach, are fat and provide lots of services, or multiple ways to do the same thing. The second-generation microkernels more follow the "pure" microkernel idea: kernels with a very small footprint, only offering the abstractions really needed, with a clean and unambigous ABI. Examples of the second generation are l4 and QNX.

[top]

The Exokernel

The Exokernel let user programs override the standard code exported by the system and the kernel itself. This leads to very fast operation , because a programmer will know how to implement the specific algorithm as fast as possible. So, exokernels let normal users take over the functionality of the kernel. But the

weakness is safety, how to restrict what user programs can do. So, this means another boom in kernel size, when protection algorithms have to be implemented. Also, exokernels use a "soft" protection boundary, trying to force the programmer use only safe languages.
This depends on the integrity of the programmers, and faulty programs can heavily weigh on the fault tolerance of the system.

[top]

Cache kernel

The cache kernel (developped at Stanford) is based on the idea of caching. It only caches threads, memory spaces, inter process communication and kernels. This cache kernel executes in kernel mode. With each cached object, an application kernel is associated. This application kernel runs in user mode and provides the management of the address spaces and thread scheduling.So, every application kernel can do thread scheduling the way it wants, e.g. the Unix scheduling and it can manage the memory how it wants.This is a very short description, but it makes clear that the Cache Kernel also offers the extensibility of exokernels.
Cache kernels provide a "hard" protection boundary: they use the kernel-mode/user-mode convention.

To my personal opinion, however, I think also well-designed microkernels can offer the extensibility needed while retaining the stablility. The big example is QNX. Most OS designers still say microkernels don't offer the performance one should want, but this is not true: a microkernel with a good design can perform as well as a monolithic OS and as the Exokernel.

[top]

Unix

Unix uses a monolithic design, implementing all services in the kernel. It offers great stability and speed, but it is technically overaged. There are off course now some enhancements to monolithic kernels, that make monolithical kernels more modern.
Modules, pieces of code that can be plugged into the kernel,offer run-time extensibility like microkernels do, but these modules run also in kernel mode, so they must be secure and stable.
Linux is off course the hype of the moment, with it's stability,it's speed and it's power, but from an architectural point of view, it is outdated.(Not the design of the part of the OS running on top of the kernel is outdated, no, Linux's (and Unix's in general) programs offer the most high-tech in the world, but the idea of a monolithical kernel itself is outdated.
Linux uses a modular design, which lets you slot in drivers and other modules into the kernel at run-time. The downside is that all drivers , even those not needing access to the hardware, run in full kernel mode and are not protected from each other and the kernel is not protected from them.

[top]

Mach

Mach was the first microkernel architecture with a lot of impact. Most modern operating systems are based on its ideas. It was developped at CMU. Mach offers everything discussed in the microkernel section. It is a microkernel of the first generation and is very large, providing more than 1 way to do something. Mach has very powerfull memory management, but this enlarges the kernel. The Mach memory management was designed to handle very large and sparse memory spaces, providing memory-mapped files etc..
Mach is a message-passing microkernel based on the client-server ideas (well, most microkernels are, but

Mach was really the big inventor).
So-called servers extend the functionality of the kernel (file systems etc). These servers can run in a seperate memory space, but also in the kernel memory space, offering bigger speed for vital services (device drivers etc). A Unix/POSIX server was thus implemented on mach, so that all Unix programs can run on Mach without needing to do lots of porting work.
All abstractions provided by the kernel are represented as objects, so Mach is fully object-oriented (allthough written completely in C). Mach uses the idea of "ports" to implement the access to these objects. Processes can send messages to objects through these ports only if they have access to them. The messages are then sent through the port to the owner of the port. So, Mach is a message-passing microkernel. Message-passing is the way the IPC works between the processes, but also how processes request services from servers and do system calls.

[top]

Windows NT

Windows NT uses a HAL that implements the hardware-dependent part of the OS. This HAL provides functions to access the hardware in a generic way.This allows for a hardware-independent kernel. The NT Hal can also emulate parts of the hardware if the actual hardware isn't available. A microkernel runs on top of this HAL, based on Mach ideas. Then, running on top of the microkernel, but also in supervisor mode, are the servers : the object manager, I/O manager, process manager, file system, GDI etc. The I/O manager contains the device drivers. These access the hardware through the HAL. All these servers, together with the microkernel, form the Windows NT executive.
The servers are running, just like the kernel, in kernel mode, so this leads to fast I/O and system services, but the downside is that all drivers have access to the whole system. What is the difference with Linux, offering as much extensibility with it's modular design?
Well, the servers/drivers in the NT executive are all running in seperate memory spaces, so they can't interfere with each other and can't play with the kernel's memory.
Windows NT is Object-oriented just like Mach. All system services and abstractions are represented as objects. Programs must have the right access rights to use them. The WindowsNT memory management is also much like that of Mach, using the same objects just with other names :-).

[top]

QNX

QNX uses a very small microkernel (only 32KB !!), that provides in-place execution, so it is suited for embedded systems too. It is very stable and the overall code size of the system is very small.This small kernel and code size allows of course for very easy development of both kernel and programs. The microkernel only implements multithreading, interrupt handling, IPC and memory management. A process manager thread (32KB) extends the kernel and runs in kernel mode too. All other drivers, servers and user programs run in user mode as normal user processes. The scheduling algorithm and overall design make it a real-time OS; when speed is your need... You can easily scale QNX from a coffee-machine to a huge SMP-server if you want. What is Microsoft talking about "scalability" ??. QNX can easily recover from faulting drivers or system processes without even rebooting and you can add and change even system DLLs (shared libraries) without rebooting. This is what we call *real* extensibility and flexibility. QNX has a really very clean design.

Visit QNX

[top]

BeOS

BeOS also uses a Mach-like microkernel. It uses pervasive multithreading, wich means it uses lots of threads, for each possible job, so it has a threaded design throughout the system. This makes BeOS very fast and very useful for multimedia applications. It is ready for multi-processing. BeOS also uses a 64-bit file system and direct graphics access, bypassing the graphical subsystem. BeOS has a simple API and it has a POSIX-interface too, with all the standard UNIX utilities and a bash shell. It is said to boot in a few seconds, unlike other operating systems.

Visit Be

[top]

L4/Fiasco

L4 is a microkernel of the second generation. It is the successor to L3. And Fiasco is a new L4-compatible kernel. The Fiasco kernel can be preempted at allmost any time, which leads to short response time5 for high-priority threads, so Fiasco is suited for real-time systems.

Visit Fiasco

[top]

Minix

Minix is a Unix-like operating system developped for learning OS design at universities. Well, actually, it is not like Unix at all. It uses a modular design, more like a microkernel. All system services are implemented as threads running on the kernel. The drivers also run as threads, but in kernel mode, statically linked. So, it's design is much more modern than Unix's, more something like NT's structure, but with the file system and user interface running in user mode. And because it's really very small, it' easy to program and teach.

[top]

Notes

1. This is not a very in-depth look to specific operating systems. Visit their web sites for more information.
2. When I write "he" in this document, I mean he or she :-)
3. If you think I made a fundamental or other error, please mail me and discuss it.

CSE 120 Principles of Operating Systems

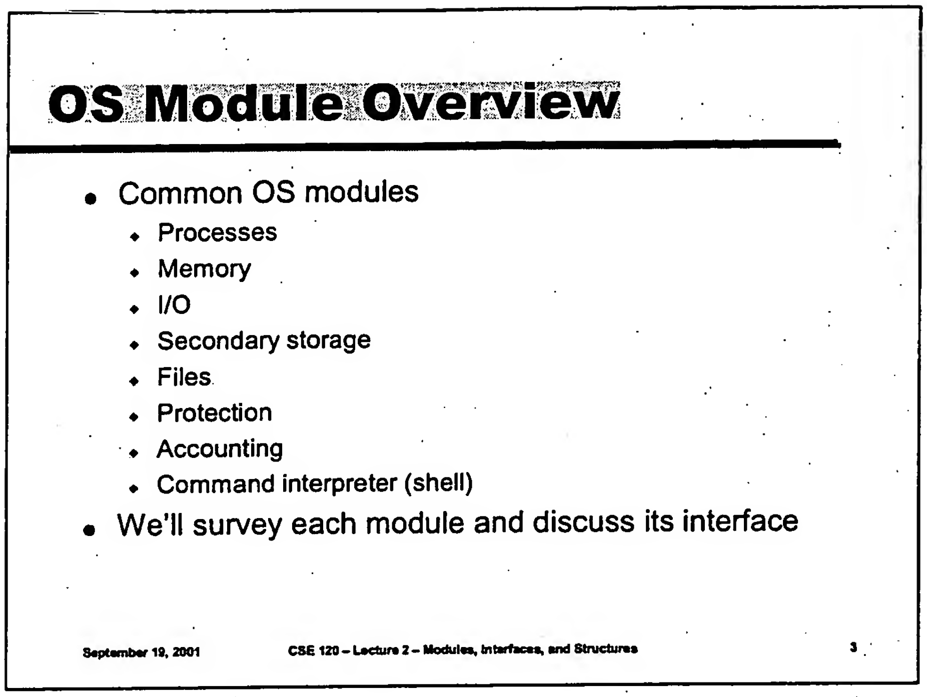
Fall 2001

Lecture 2: Operating System Modules, Interfaces, and Structures

Geoffrey M. Voelker

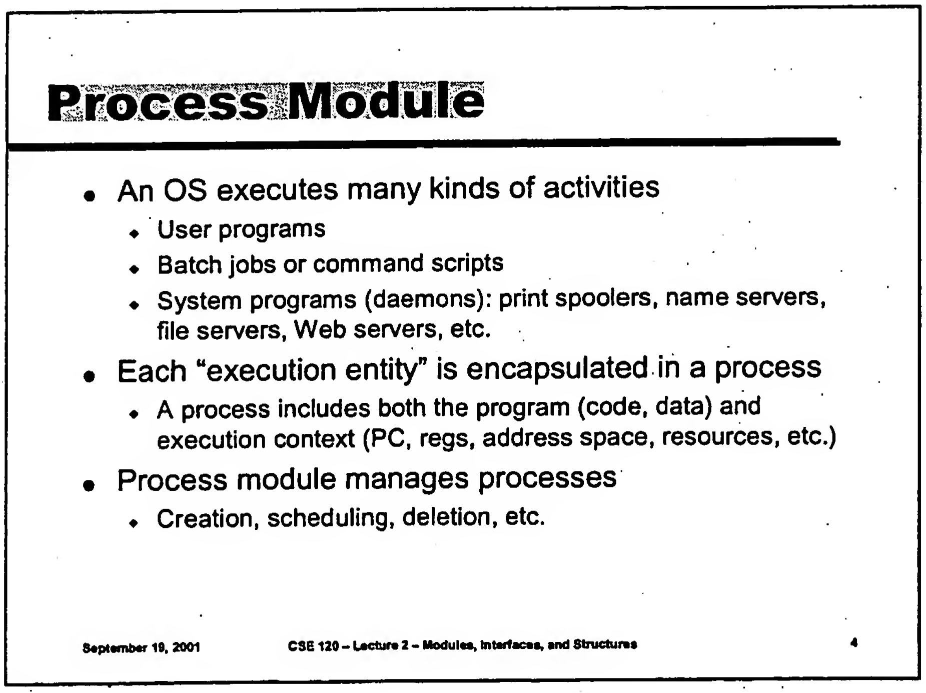
Modules, Interfaces, Structure

- We roughly defined an OS as the layer of software between hardware and applications
- Now we're going to survey the support OSes provide to applications
  - Modules – OS services and abstractions
  - Interfaces – operations supported by components
  - Structure – how components get hooked together



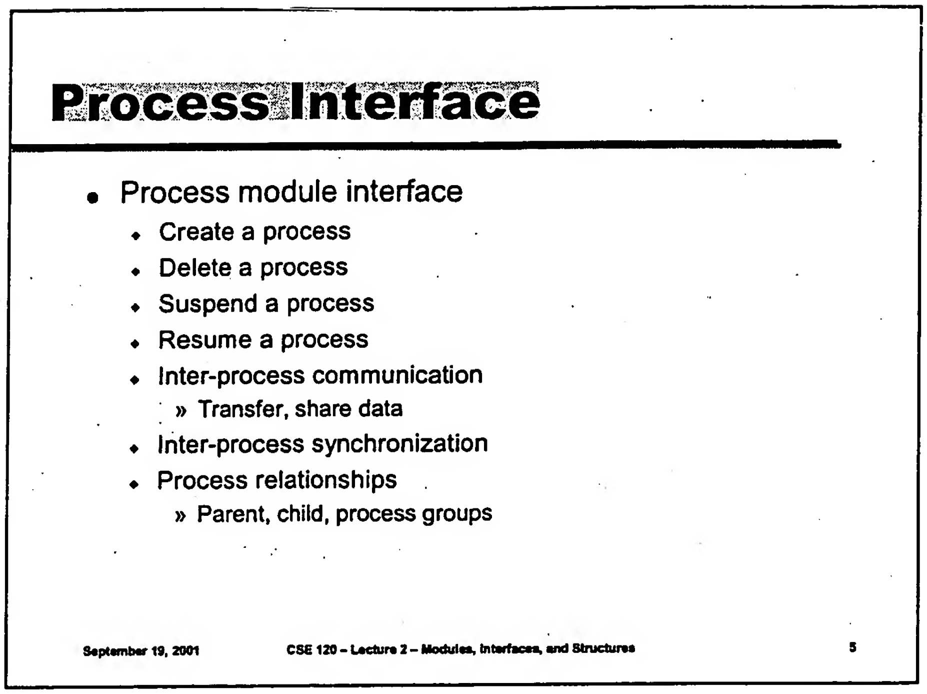
OS Module Overview

- Common OS modules
  - Processes
  - Memory
  - I/O
  - Secondary storage
  - Files
  - Protection
  - Accounting
  - Command interpreter (shell)
- We'll survey each module and discuss its interface



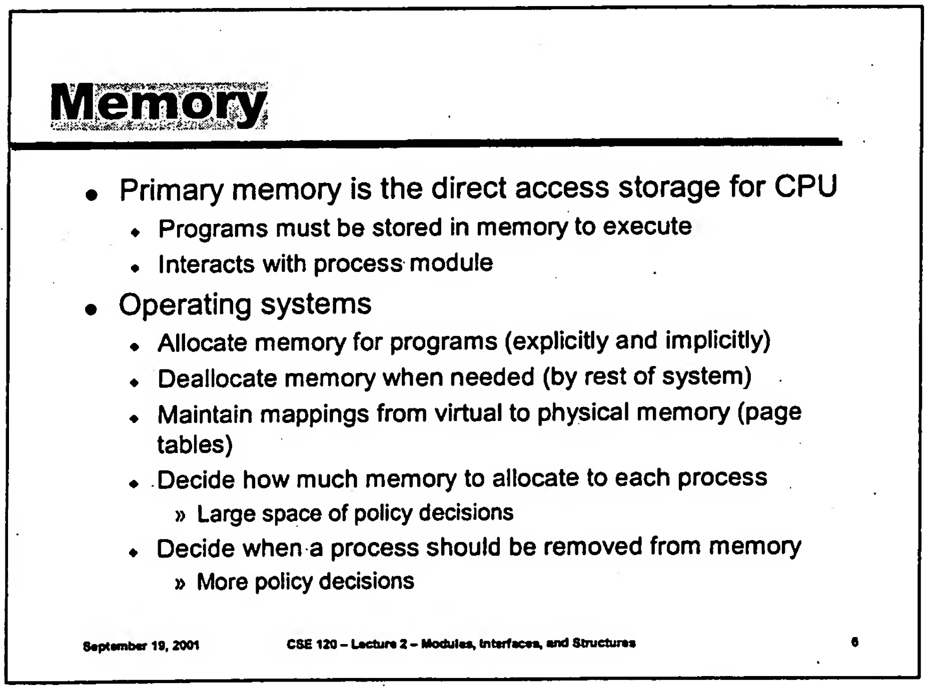
Process Module

- An OS executes many kinds of activities
  - User programs
  - Batch jobs or command scripts
  - System programs (daemons): print spoolers, name servers, file servers, Web servers, etc.
- Each "execution entity" is encapsulated in a process
  - A process includes both the program (code, data) and execution context (PC, regs, address space, resources, etc.)
- Process module manages processes
  - Creation, scheduling, deletion, etc.



Process Interface

- Process module interface
  - Create a process
  - Delete a process
  - Suspend a process
  - Resume a process
  - Inter-process communication
    - » Transfer, share data
  - Inter-process synchronization
  - Process relationships
    - » Parent, child, process groups



Memory

- Primary memory is the direct access storage for CPU
  - Programs must be stored in memory to execute
  - Interacts with process module
- Operating systems
  - Allocate memory for programs (explicitly and implicitly)
  - Deallocate memory when needed (by rest of system)
  - Maintain mappings from virtual to physical memory (page tables)
  - Decide how much memory to allocate to each process
    - » Large space of policy decisions
  - Decide when a process should be removed from memory
    - » More policy decisions

I/O

- Much of an OS deals with device I/O
  - One of the main reasons we use OSes
  - Hundreds of thousands of lines of code in NT for I/O, drivers
- The OS provides a standard interface between programs (user or system) and devices
  - File system (disks), sockets (network), frame buffer (video)
- Device drivers are the routines responsible for controlling I/O devices
  - OS defines an interface for each class of devices (e.g., disks)
  - A driver implements interface, encapsulates device-specific knowledge (initiation and control, interrupt handling, errors)

Secondary Storage

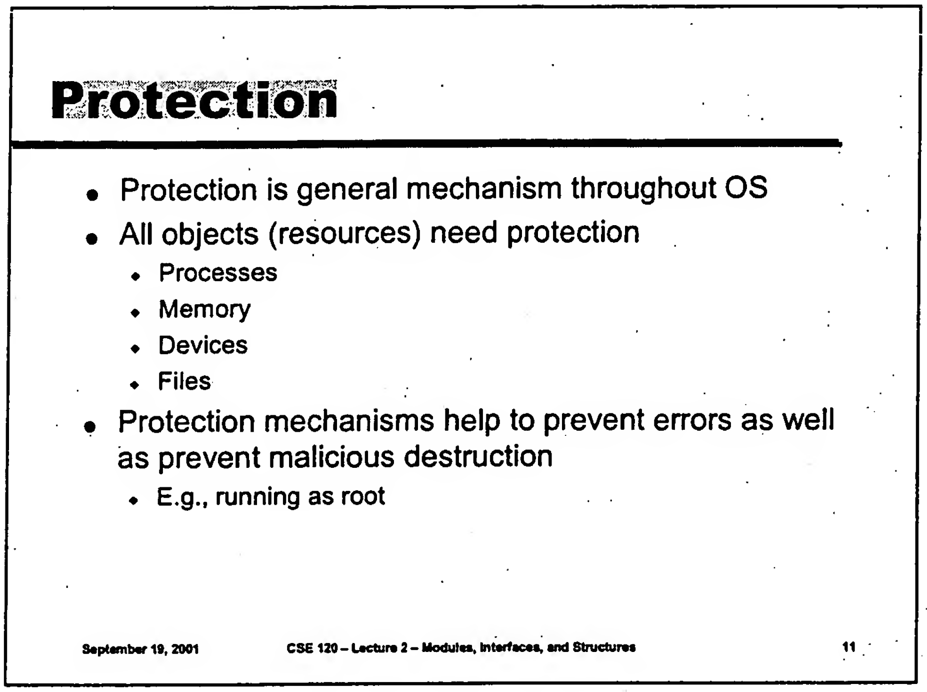
- Secondary storage (disk) is the persistent memory
  - It endures system failures (for the most part)
- Low-level OS routines are often responsible for low-level disk functions
  - Read/write blocks
  - Schedule requests (optimize arm movement)
  - Device errors
- Usually independent of file system
  - Although there might be cooperation (e.g., free space management)
  - Low-level knowledge can help FS performance (placement)

File System

- Secondary storage devices are too crude to use directly for long-term storage
  - Read/write physical device blocks too low-level for programs
- The file system provides a much higher level, more convenient abstraction for persistent storage
  - Objects (files, directories) and interfaces (read, write, etc.)
- Files are the basic storage entity
  - A file is a named collection of persistent information
- Directories are special files that contain the names of other files + metadata (data about files, attributes)
  - Directories have all properties of files ("inheritance")

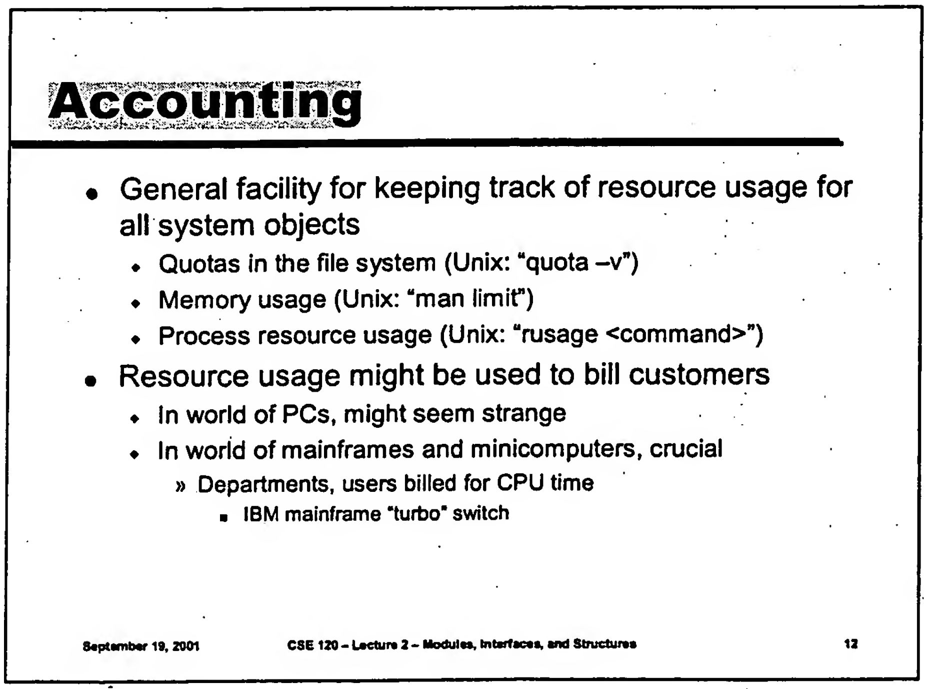
File System Interface

- File system interface provides standard file operations
  - Existence: File/directory creation, deletion
  - Manipulation: open, read, write, append, rename, close, etc.
  - Sometimes higher-level operations
    - » File copy, change notification (NT)
    - » Records (IBM)
- File system also provides general services
  - Backup
  - Consistency
  - Accounting and quotas



Protection

- Protection is general mechanism throughout OS
- All objects (resources) need protection
  - Processes
  - Memory
  - Devices
  - Files
- Protection mechanisms help to prevent errors as well as prevent malicious destruction
  - E.g., running as root



Accounting

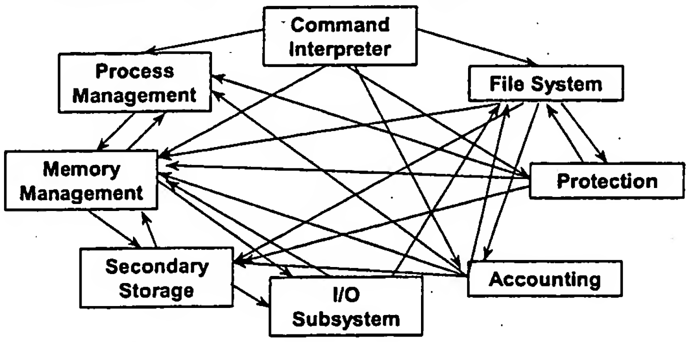
- General facility for keeping track of resource usage for all system objects
  - Quotas in the file system (Unix: "quota –v")
  - Memory usage (Unix: "man limit")
  - Process resource usage (Unix: "rusage <command>")
- Resource usage might be used to bill customers
  - In world of PCs, might seem strange
  - In world of mainframes and minicomputers, crucial
    - » Departments, users billed for CPU time
      - IBM mainframe "turbo" switch

Command Interpreter (Shell)

- Process that handles
  - Handles user input (commands)
  - Manages subprocesses
  - Executes script files (files of commands)
- On some systems, CI is part of OS
  - Users constrained to use that CI (DOS)
- Others, it is just another user-level process
  - Unix shell
  - Any program can be a CI (sh, csh, ksh, bash, etc.)
- Or, there may not be a command language at all
  - MacOS (hey, where's the shell?)

The Challenge of Structure

- It is clear what modules an OS should provide
- Not so clear how to hook them together (well)...

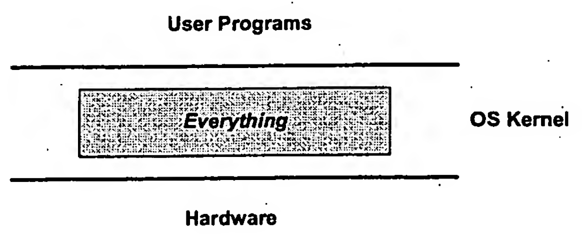


OS Structure

- An OS consists of all of the above modules, others we haven't discussed, system programs (privileged and non-privileged), etc.
- The challenge
  - How do we organize it all?
  - What are the modules, where do they exist?
  - How do they cooperate?
- How do we build a complex software system that is:
  - Large and complex
  - Concurrent
  - Long-lived and evolving
  - Performance-critical

Monolithic Kernels

- Traditionally, OSes such as Unix are built as a monolithic kernel (BSD, SYSV, Solaris, Linux, etc.)

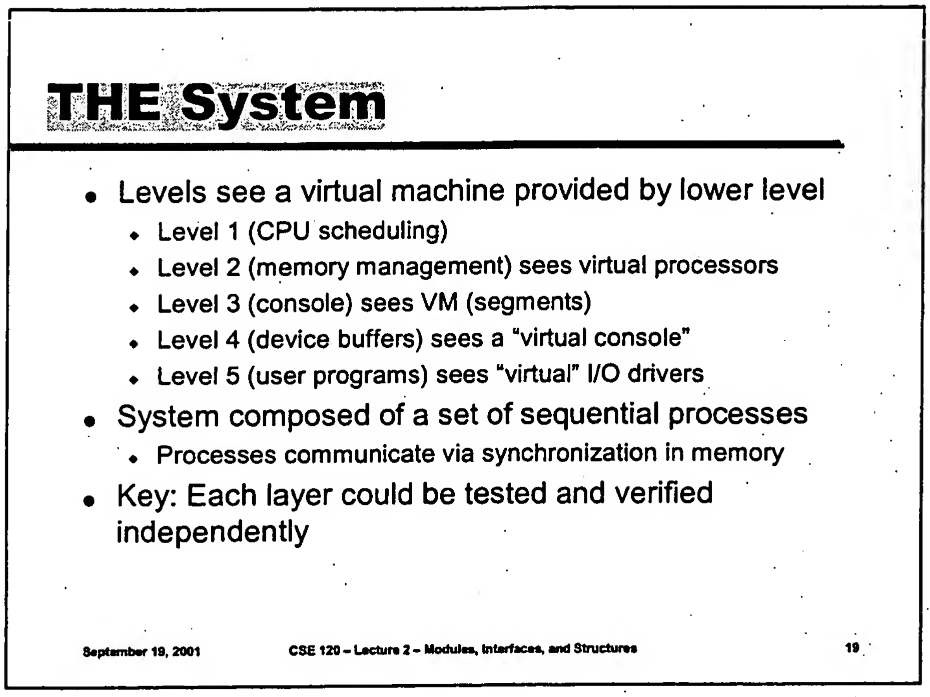


Monolithic Problems

- Monolithic kernels have one great feature
  - The overhead of inter-module interaction is low
- Monolithic kernels have a number of problems
  - Hard to understand
  - Hard to modify
  - Hard to maintain
  - Unreliable (a bug in one module can take down entire system)
- From the beginning, OS designers have sought ways to organize the OS to simplify design and construction

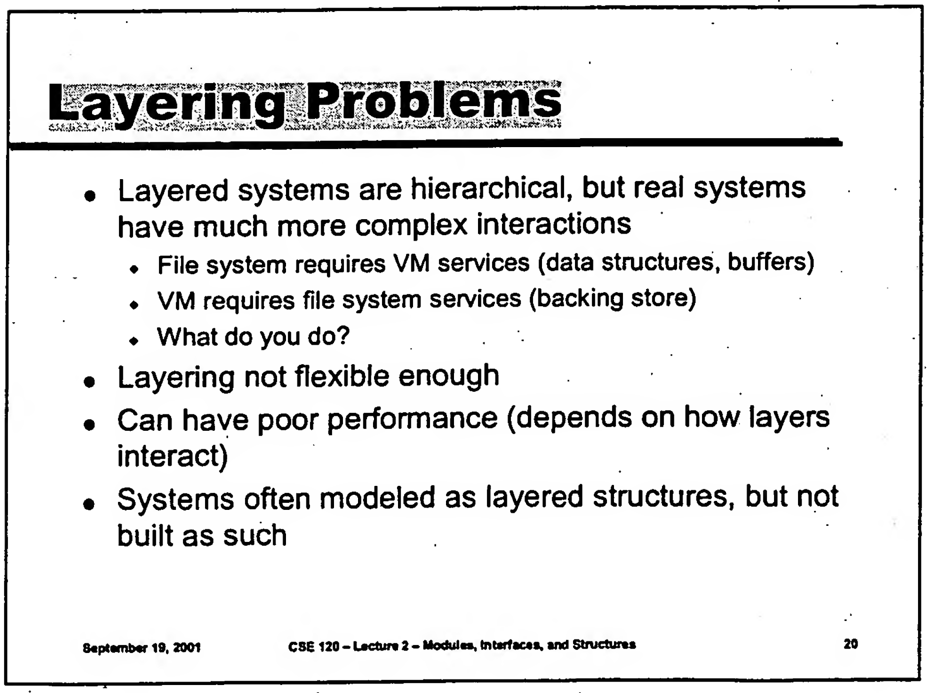
Layering

- An early approach is layering
  - Each system service is a layer
  - Each layer defines and implements an abstraction for the layer above it
  - Layers in effect "virtualize" the layer below
- Approach first used by Dijkstra's THE system (1968)
  - Analogous to protocol layers in networking stacks



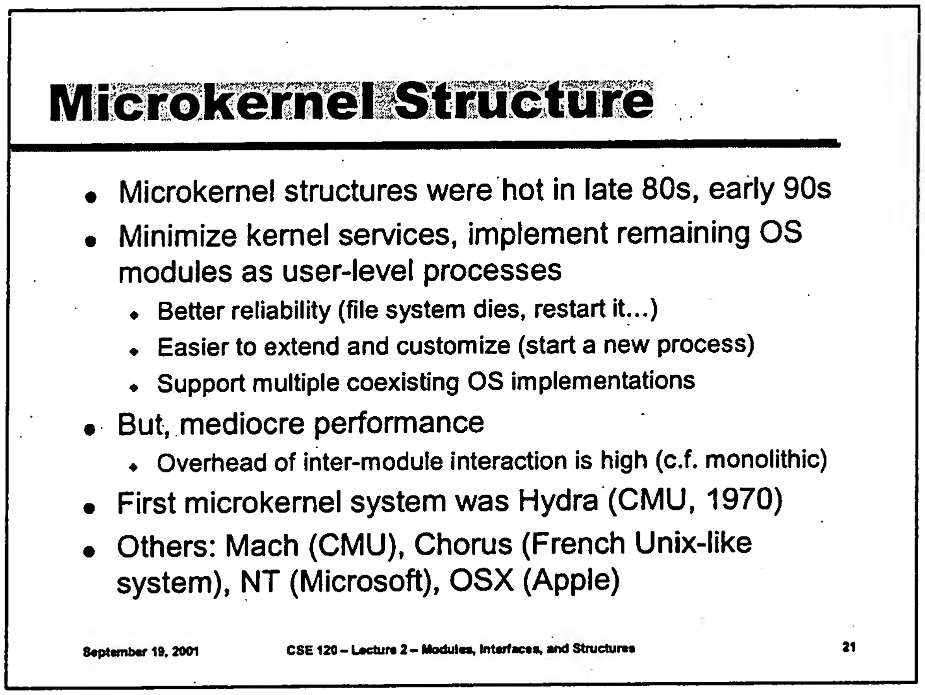
THE System

- Levels see a virtual machine provided by lower level
  - Level 1 (CPU scheduling)
  - Level 2 (memory management) sees virtual processors
  - Level 3 (console) sees VM (segments)
  - Level 4 (device buffers) sees a "virtual console"
  - Level 5 (user programs) sees "virtual" I/O drivers
- System composed of a set of sequential processes
  - Processes communicate via synchronization in memory
- Key: Each layer could be tested and verified independently



Layering Problems

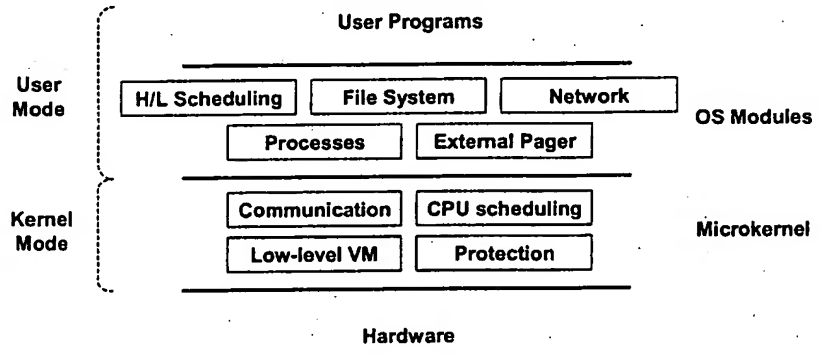
- Layered systems are hierarchical, but real systems have much more complex interactions
  - File system requires VM services (data structures, buffers)
  - VM requires file system services (backing store)
  - What do you do?
- Layering not flexible enough
- Can have poor performance (depends on how layers interact)
- Systems often modeled as layered structures, but not built as such



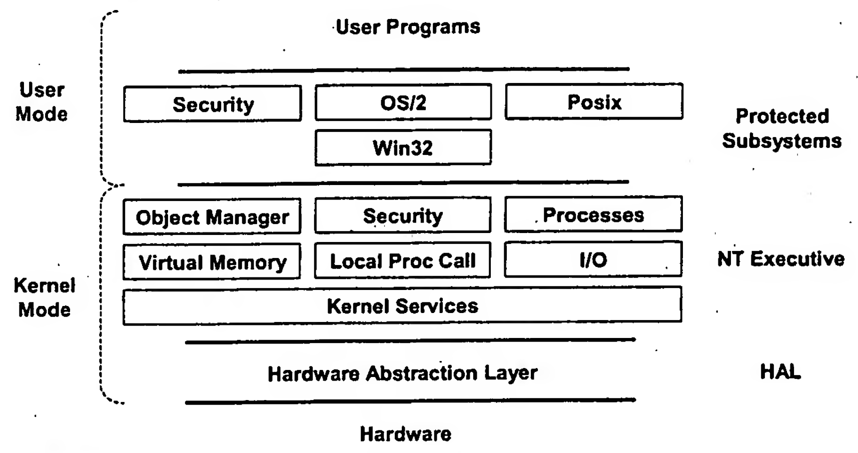
Microkernel Structure

- Microkernel structures were hot in late 80s, early 90s
- Minimize kernel services, implement remaining OS modules as user-level processes
  - Better reliability (file system dies, restart it...)
  - Easier to extend and customize (start a new process)
  - Support multiple coexisting OS implementations
- But, mediocre performance
  - Overhead of inter-module interaction is high (c.f. monolithic)
- First microkernel system was Hydra (CMU, 1970)
- Others: Mach (CMU), Chorus (French Unix-like system), NT (Microsoft), OSX (Apple)

Microkernel Structure



Windows NT 3.1 (1993)



Next time...

- Read Chapter 12

3. Microkernel-based OS Efforts

So what is a Microkernel and why would you run Unix on top of another OS?

A Microkernel is a highly modular collection of powerful OS-neutral abstractions, upon which can be built operating system servers. In MACH , these abstractions (tasks, threads, memory objects, messages, and ports) provide mechanisms to manage and manipulate virtual memory (VM), scheduling, and interprocess communication (IPC).

This modularity enables scalability, extensibility, and portability not typically found in monolithic or conventional operating systems. Because the MACH microkernel is OS-neutral, different OS ``personalities'' such as Linux can be hosted on the microkernel. Perhaps the most significant advantage is that there is a focus on portability in the microkernel design and in decoupling that from the hosted OS.

Microkernels move many of the OS services into ``user space'' that on other operating systems are kept in the kernel. This has significant effects in the following areas:

- Robustness

  If there is a problem with a particular service, it normally can be reconfigured and restarted without having to restart the complete OS. This should be helpful for situations requiring high availability.

  Moreover, since services would now run in completely independent memory spaces (which is *not* the case for kernel-level services), bugs and misconfiguration will be less able to corrupt the kernel.

  Moreover, the ``true kernel'' winds up being smaller in scope, and thus ought to be easier to understand and verify. The L4 microkernel occupies about 32K of memory, which severely limits how complex it can realistically be.

- Security

  Services that run within the kernel effectively have ``ring 0'' privileges, which is to say that they can do anything, anywhere, at any time. Unix processes that run as *root* don't have that much control over the system.

  A problem at present with Linux is that any program that has graphics access must be setuid root because that is the only way of having permission to access the hardware. This has the unfortunate effect that the program gets ``root'' access to *everything* on the system, and not just the screen.

  By running the services as lower level user processes, their access to system resources (*e.g.* - the ability to mess things up) is far more restricted.

  Moreover, ``security'' becomes less monolithic, which allows even system services, and indeed security services for that matter, to themselves be forced to comply with security requirements.

- Configurability

  Services can be changed without need to restart the whole system.

Work has been ongoing to (for instance) make Linux more dynamically re-configurable, loadable kernel modules being a good example.

- Makes Coding Easier

  Kernel code usually requires the use of special memory allocation and output routines since the kernel cannot depend on a lower level to manage these things for it. User-mode code is thus simpler to write than ``kernel" code, because it doesn't need to worry about kernel-specific restrictions.

- Lower ``fixed" memory footprint.

  Kernel-allocated memory (code+data) is generally strictly forced to stay in memory; swapping it out is forbidden. The more ``kernel-like" code that is moved to user space, the more of the services of the OS can be swapped out if they are used infrequently.

  This is the reason why it is advantageous for Linux to run X as a separate process rather than throwing it into the kernel. Portions that are not being used can get swapped out.

- Moves us more towards Real Time performance.

  Interrupts are usually turned off when in kernel mode to prevent interruption of critical processing. The less code in the kernel, the less that ``interrupts are interrupted."

  Processes with Real-Time requirements would be given better access to the microkernel than other processes, including some that would normally be considered 'base' OS services.

- Simplifies Symmetric Multiprocessing code

  Managing multiple processors requires a fair bit of internal bookkeeping. The more services that sit in the kernel, the more bookkeeping work code is required, and the less effective use can be made of the additional processors. Changing a large, single-threaded monolithic kernel to make effective use of SMP hardware is very difficult.

  If the OS runs atop a microkernel specifically written to make effective use of SMP, then SMP support does not need to be obtrusive in the ``main" kernel.

- Factors out complexity

  Software complexity that relates to (for instance) hardware interfacing can be factored out into clearly separate modules. This allows the overall system functionality and complexity to grow further without becoming completely unmanageable.

The microkernel approach has been taken by Apple in creating a version of Linux that runs as a ``personality" on top of MACH on PowerMacs. This should make it considerably easier to port Linux to additional system architectures, as MACH is considerably smaller than Linux.

Are there any downsides to the use of microkernels? Yes, indeed.

- Most microkernels are not tiny, despite the name. (QNX being a notable exception; L4 possibly being an exception.)

  The overall RAM footprint of the system will likely increase.

- Communications between components of the extended ``OS" requires that formalized message-passing mechanisms be used.

- As a result, code must be written to use the formal mechanisms, rather than processes being able to informally use system memory. This may reduce performance.

  Analysis of the HP PA-RISC port of Linux atop MACH indicates that the microkernelled version of Linux runs approximately 10% slower than HP/UX.

- New kinds of deadlocks and other error conditions are possible between system components that would not be possible with a monolithic kernel.

Probably the fundamental consideration encouraging microkernel development is the growth of interest in SMP and other multiprocessing applications. Simplifying ports to new architectures and encouraging compatibility with other OSes being close behind. And then there's the Hurd arguments...

Discussions of implementing Linux over a microkernel come up periodically in the Linux kernel/system design newsgroup; MkLinux represents some concrete steps in this direction. Many of these ideas can be applied to a non-microkerneled OS in at least limited ways; Linux ``kernel modules'' being a good case in point.

- Mach

  The Mach home page at Carnegie Mellon University.

- Lites

  Lites is a 4.4 BSD Lite based server and emulation library that provides free Unix functionality to a MACH based system. Probably most reminiscent of running an OS that feels like NetBSD atop MACH .

- GNU Hurd

  The kernel for the ``official'' FSF/GNU operating system project.

- xMach.org

  A (seemingly defunct) site seeking to support continuing development of Mach4 and Lites1.

- dmoz.org - Microkernel OSes

- Argante Virtual Operating System

  It provides a virtual environment for running applications atop a Unix system. Its design is informed by those of QNX, Hurd, Inferno, and such.

  It consists of a low level virtual machine that is treated like a microkernel . Operations are implemented using loadable modules that run atop that kernel.

  Not unlike the Java virtual machine or P-Code concepts, this provides its own low level, hardware independent "machine language."

  Unlike Unix, the Argante notion of "process" is not something spawned/killed, but is rather more like the VMS/MVS notion of functional services that are started/stopped from the console.

- L4

A small microkernel (most of them are pretty large) oriented towards message-passing. One of the major applications involves running Linux top L4, in the Linux on L4 project.

There is a project to put Hurd atop L4.

- FIASCO: L4-compatible Microkernel

  The people who built L4 have also produced FIASCO, a kernel *available under the GPL* that is compatible with L4... Being mostly written in C++, it is slower than L4, but more readily ported.

- Mungi - another OS that runs atop L4

  It is *not* a Unix-like system; it implements a single 64 bit address space, shared by all processes and processors.

- Jaluna

  From the makers of Chorus, the C++-based microkernel OS...

3.1. Another Comparison of Microkernels versus Other Architectures

The fundamental attribute that distinguishes monolithic vs. microkernel vs. exokernel architectures is what the architecture implements in kernel space (that which runs in supervisor mode on the cpu) vs. what the architecture implements in user space (that which runs in non-supervisor mode on the cpu).

The monolithic architecture implements all operating system abstractions in kernel space, including device drivers, virtual memory, file systems, networking, device/cpu multiplexing, etc.

The microkernel architecture abstracts lower-level os facilities, implementing them in kernel space, and moves higher-level facilities to processes in user space. Usually what distinguishes higher vs. lower-level os facilities is that which can be implemented in a platform independent manner vs. that which cannot. Following from that another distinguishing characteristic is what is sufficiently general to provide for various operating-system ``personalities'' and what is not. In microkernel architectures device-drivers, virtual memory, process/task/thread management/scheduling, and other such facilities are implemented in the kernel, and parallel facilities that specialize those facilities for the operating system's personality are implemented in user-space processes. Also implemented in user space are file systems, networking, etc. that employ the lower-level facilities provided by the microkernel (like device drivers).

In contrast, the exokernel architecture implements nothing in kernel space. The exokernel's sole purpose is to securely multiplex hardware resources among user-space processes. Device drivers, virtual memory, even cpu multiplexing and process management are implemented in user space. Supervisor-mode hardware events, like timer ticks, page faults, etc., activate stub handlers in the kernel that simply pass the event to a user-level process that implements the relevant facility's policy. The same system can simultaneously implement forward and inverted page tables, compute-job-friendly or interactive-job-friendly process scheduling, and an application can pick and choose which ever policies will provide it with the best performance.

The exokernel architecture is essentially the extension of the philosophy of RISC cpu architecture to the operating system level. The only exokernel architecture that I know of (MIT Aegis) has come up with some very novel ways to implement this.

Were I to implement my dream system it would be of exokernel architecture.
-- On 17 Mar 1998 21:12:52 GMT, <brianm@csa.bu.edu> Brian Mancuso wrote:

3.2. The Fundamental Challenge of Microkernels

In a 2003 discussion on alt.os.multics , there was some disagreement between Linus Torvalds and some of the Multicians. One of the very best comments was the following...

> I can understand you pet peeve about microkernels and message passing when looking at Mach or Minix (not as well when looking at QNX , which performs quite well, and isn't bloated). On the other hand, a lot of "system" work on Linux gets done by efficient message passing; it's just a part of the system you are not involved in at all (hint: it's the X Window System). Why does message passing work quite well between X and the client? Because X manages to pack a lot of messages together before it actually switches tasks. That's because X's calls are asynchronous (with a few exceptions, some of them mistakes like `XInternAtom()`).
>
> Unix's syscalls all are synchronous. That makes them a bad target for a microkernel, and the primary reason why Mach and Minix are so bad - they want to emulate Unix on top of a microkernel. Don't do that.
>
> If you want to make a good microkernel, choose a different syscall paradigm. Syscalls of a message based system must be asynchronous (*e.g.* asynchronous IO), and event-driven (you get events as answers to various questions, the events are in the order of completion, not in the order of requests). You can map Unix calls on top of this on the user side, but it won't necessarily perform well.
>
> --Bernd Paysan

This seems pretty convincing. The L4 designers found indeed that message passing overhead led to their microkernels operating more slowly than monolithic kernels, *running the same Unix APIs*. If applications were redesigned to use asynchronous message passing, they might perform very well, but when they use the synchronous Unix APIs, performance *will suffer*.

This is also consistent with embedded applications running on QNX performing well: if you design the application to use its message passing APIs, it *will* work well.

3.3. Exokernels

Exokernels are a further extension of the microkernel approach where the "kernel" per se is almost devoid of functionality; it merely passes requests for resources to "user space" libraries.

This would mean that (for instance) requests for file access by one process would be passed by the kernel to the library that is directly responsible for managing file systems. Initial reports are that this in particular results in significant performance improvements, as it does not force data to even pass through kernel data structures.

They are presently still largely in "research mode;" there are not, at this time, any readily available implementations that one could run at home.

- A Caching Model of OS Kernel Functionality
- FreeMWare.org
- Summary of Exokernel: An Operating System Architecture for Application-Level Res. mgmt
- MIT Exokernel Operating System (Paper)

- Application Performance and Flexibility on Exokernel Systems (Paper)
- exopc Distribution (MIT Exokernel)
- Adrenaline OS

  An OS based on the MIT Exokernel.

Prev	Home	Next
Terry Lambert's Top Ten List For A Non-Proprietary OS	Up	GNU Hurd

Google [Google Search]
[✓ Search WWW] Search cbbrowne.com Search ntlug.org

[Report Broken Links]

If this was useful, let others know by an Affero rating

amazon.com. Privacy Information

Proceedings of the Useni...
Usenix Assoc

Proceedings of the Useni...
Usenix Assoc

Contact me at cbbrowne@acm.org